

Filtros espaciais

[Nota para mim: Preciso reorganizar esta apostila. Trocar exemplo 1D por template maior. Mostrar que quanto mais parecido a instância de A com Q, maior a correlação em P.]

As imagens usadas nesta aula estão em:

<http://www.lps.usp.br/hae/apostila/filtconv.zip>

e podem ser baixadas em Linux com os comandos:

```
Linux$ wget -U 'Firefox/50.0' http://www.lps.usp.br/hae/apostila/filtconv.zip
Linux$ unzip filtconv.zip
```

1. Introdução

Um filtro espacial (ou filtro janela móvel ou operador restrito à janela) é uma transformação de imagem onde a cor de um pixel da imagem de saída é escolhida em função das cores da vizinhança (janela) desse pixel na imagem de entrada (figura 1). Os filtros desempenham funções essenciais em diversas áreas do processamento de imagens. Os filtros comumente utilizados em processamento de imagens incluem:

- Filtros lineares espaciais (convolução): média móvel, filtro gaussiano, filtro laplaciano, gradiente, convolução, correlação, etc.
- Filtros baseados em estatística de ordem (ou rank filter ou percentile filter) [Singhaniya2021]: filtro mediano, filtro do ponto médio, filtro média α -trimmed.
- Filtros morfológicos: erosão, dilatação, abertura, fechamento, hit-miss, etc.
- Difusão anisotrópica, total variation minimization.

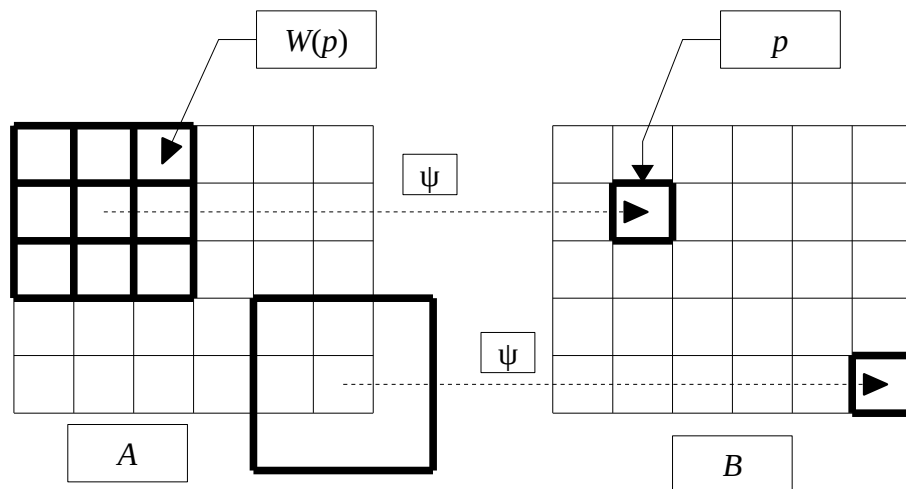


Figura 1: Um filtro Ψ decide a cor de um pixel p na imagem de saída B analisando uma vizinhança $W(p)$ do pixel p na imagem de entrada A dentro da janela W . Modo “same” faz o tamanho da imagem de saída ser igual ao de entrada.

Um filtro espacial Ψ escolhe a cor de um pixel p da imagem de saída B baseado nas cores da vizinhança W de p na imagem de entrada A . Ou seja, Ψ é definido através de uma janela W e uma função característica ψ que transforma os valores dos pixels dentro da janela W na cor de saída:

$$W = (W_1, W_2, \dots, W_w), W_i \in Z^2 \quad \text{e} \quad \psi: L^w \rightarrow L \quad (L \text{ é o espaço das cores})$$

tal que

$$B(p) = \Psi(A)(p) = \psi(A(W_1 + p), A(W_2 + p), \dots, A(W_w + p)), \quad \forall p \in Z^2.$$

O tamanho da imagem de saída de um filtro varia de acordo com o “modo”. Matlab chama os 3 modos diferentes de aplicar filtros (que resultam em saídas de tamanhos diferentes) de “same”, “valid” e “full”. Esta mesma denominação é utilizada em várias outras bibliotecas, incluindo Keras/Tensorflow.

A figura 1 mostra o modo “same”, onde a imagem de saída B tem o mesmo tamanho da entrada A . Neste caso, a janela pode estar parcialmente fora do domínio de A (janela inferior direita). Se isso acontecer, o filtro deve escolher algum método para calcular o valor do pixel de B mesmo com alguns pixels fora do domínio de A . Por exemplo, é possível considerar que os pixels fora do domínio são todos zeros, ou aproximar os pixels fora do domínio para o pixel dentro do domínio espacialmente mais próximo, ou adotar alguma outra estratégia especial.

A figura 2 mostra o modo “valid”, onde a imagem de saída é menor que a imagem de entrada. A imagem de saída contém somente os pixels p onde a janela correspondente $W(p)$ na imagem de entrada A cabe inteiramente dentro do domínio.

O modo “full” é pouco usado e não explicarei aqui.

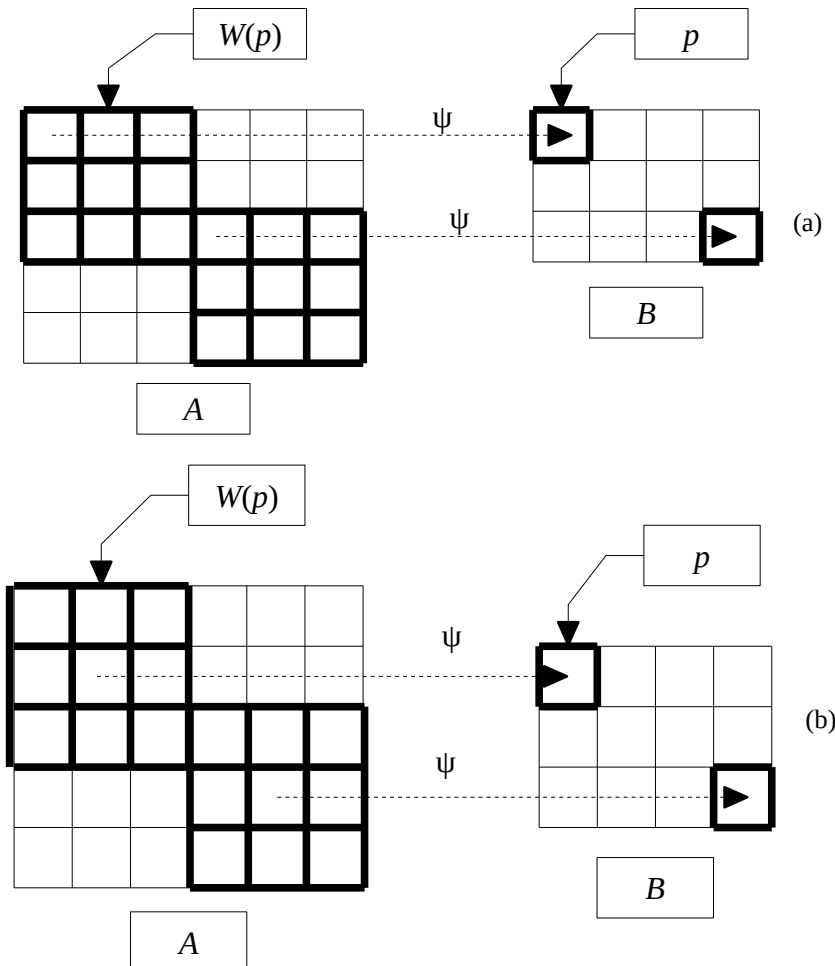


Figura 2: Filtro aplicado no modo “valid”, onde a imagem de saída é menor que a entrada. (a) É possível imaginar que um pixel de saída corresponde ao canto superior esquerdo da janela. (b) É possível imaginar que um pixel de saída corresponde ao centro da janela.

Nota: Como veremos, rede neural convolucional é uma “mistura” da convolução (filtro linear) com rede neural. Uma rede neural convolucional utiliza um neurônio para calcular a função característica ψ .

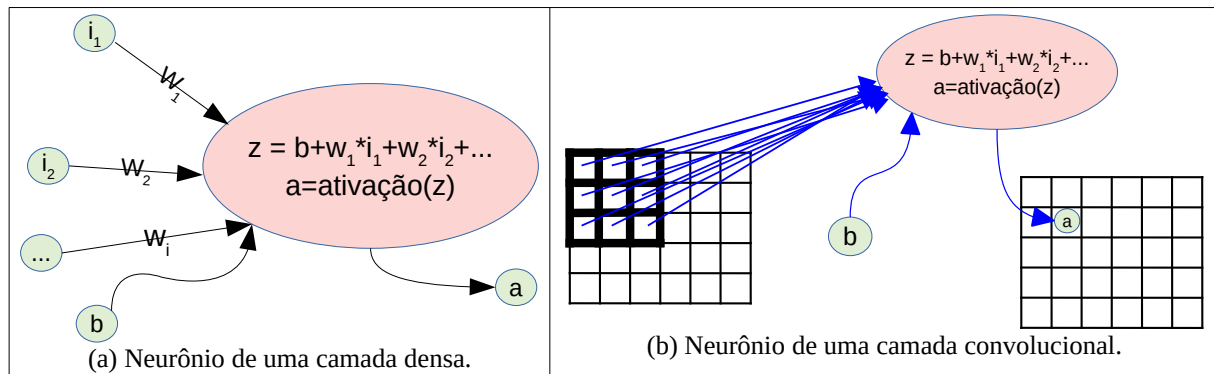


Figura F1: Neurônios de camada densa e de camada convolucional.

2. Filtro média móvel (box kernel)

Vamos estudar primeiro um filtro bem simples: média móvel. Este filtro pertence à classe de filtros lineares e calcula a média aritmética dos pixels dentro da janela. O programa 1 implementa esse filtro, usando janela 3×3 . A figura 3 mostra as imagens entrada e saída deste programa. Observamos que a imagem de saída ficou “borrada”.

```

1 //media_borda.cpp - 2024
2 #include <opencv2/opencv.hpp>
3 using namespace std;
4 using namespace cv;
5
6 Mat_<uchar> mediamov(Mat_<uchar> a) {
7     Mat_<uchar> b(a.rows,a.cols,uchar(128));
8     for (int l=1; l<b.rows-1; l++)
9         for (int c=1; c<b.cols-1; c++) {
10             int soma=0;
11             for (int l2=-1; l2<=1; l2++)
12                 for (int c2=-1; c2<=1; c2++) {
13                     int l3=l+l2; int c3=c+c2;
14                     soma = soma+a(l3,c3);
15                 }
16             b(l,c) = round(soma/9.0);
17         }
18     return b;
19 }
20
21 int main() {
22     Mat_<uchar> a=imread("lion.png",0);
23     Mat_<uchar> b=mediamov(a);
24     imwrite("media_borda.png",b);
25 }

```

Programa 1: Filtro média móvel 3×3 sem processar borda.

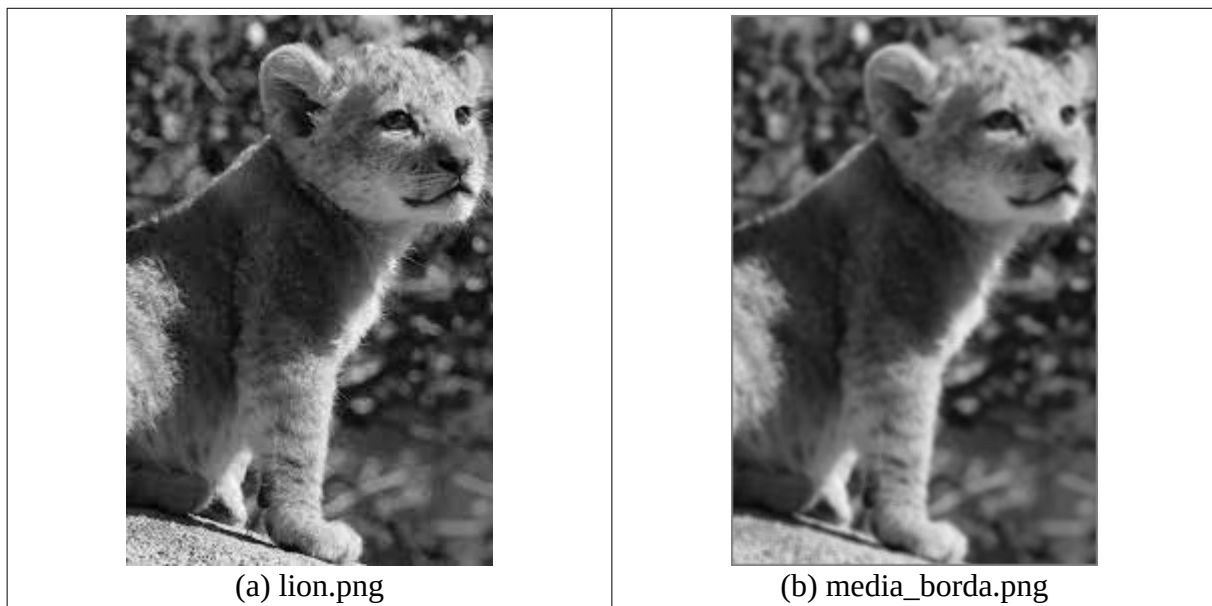


Figura 3: Entrada e saída do filtro móvel 3×3. A saída está borrada. Não foram processadas a primeira e a última linha/coluna, que ficaram cinzas.

Para não acessar pixels fora do domínio da imagem de entrada, a primeira e a última linha/coluna não foram processadas e deixadas com a cor cinza (verifique o início e o fim dos loops *for* nas linhas 8 e 9 do programa 1).

Para processá-las, precisamos definir o que fazer quando algum pixel fora do domínio da imagem cair dentro da janela. Por default, OpenCV adota a estratégia “BORDER_REFLECT_101”:

g fedcb | abcdefgh | g fedcba

https://docs.opencv.org/4.x/d2/de8/group_core_array.html#ga209f2f4869e304c82d07739337eae7c5

Isto é, a coluna índice -1 é igual à coluna 1; coluna -2 é igual à coluna 2; etc. Se a imagem tem 8 colunas, coluna índice 8 é igual a coluna 6; coluna 9 é igual a coluna 5; etc. O mesmo vale para as linhas. Os comandos abaixo permitem usar esta condição de borda.

```
if (l3<0) l3=-l3;
if (a.rows<=l3) l3=a.rows-(l3-a.rows+2);
if (c3<0) c3=-c3;
if (a.cols<=c3) c3=a.cols-(c3-a.cols+2);
```

Programa 2 implementa média móvel com esse tratamento de borda. Executando esse programa na imagem lion.png (figura 4a) obtemos a figura 4b sem as bordas cinzas.

Agora, vamos aplicar esse mesmo filtro na imagem ruído.png (figura 4c). Obtemos a figura 4d. O filtro média móvel “espalhou” os ruídos mas não os eliminou.

Programa 3 é a tradução para Python do programa 2.

Ao calcular a média aritmética dos pixels dentro da janela, precisamos tomar cuidado para efetuar divisão em ponto flutuante (e não calcular o quociente da divisão de inteiro). Para

isso, escrevemos “9.0” em vez de “9” em C++ e usamos o operador “/” em vez de “//” em Python.

```
1 //media.cpp - 2024
2 #include <opencv2/opencv.hpp>
3 using namespace std;
4 using namespace cv;
5
6 Mat_<uchar> mediamov(Mat_<uchar> a) {
7     Mat_<uchar> b(a.rows,a.cols);
8     for (int l=0; l<b.rows; l++)
9         for (int c=0; c<b.cols; c++) {
10             int soma=0;
11             for (int l2=-1; l2<=1; l2++)
12                 for (int c2=-1; c2<=1; c2++) {
13                     int l3=l+l2; int c3=c+c2;
14                     if (l3<0) l3=-l3;
15                     if (a.rows<=l3) l3=a.rows-(l3-a.rows+2);
16                     if (c3<0) c3=-c3;
17                     if (a.cols<=c3) c3=a.cols-(c3-a.cols+2);
18                     soma = soma+a(l3,c3);
19                 }
20             b(l,c) = round(soma/9.0);
21         }
22     return b;
23 }
24
25 int main() {
26     Mat_<uchar> a=imread("lion.png",0);
27     Mat_<uchar> b=mediamov(a);
28     imwrite("media.png",b);
29 }
```

Programa 2: Filtro média móvel 3×3 processando as bordas.

```
1 #media-py.py 2024
2 import cv2
3 import numpy as np
4
5 def mediamov(a):
6     b=np.empty(a.shape)
7     for l in range(a.shape[0]):
8         for c in range(a.shape[1]):
9             soma=0
10             for l2 in range(-1,2):
11                 for c2 in range(-1,2):
12                     l3=l+l2; c3=c+c2;
13                     if l3<0: l3=-l3;
14                     if a.shape[0]<=l3: l3=a.shape[0]-(l3-a.shape[0]+2);
15                     if c3<0: c3=-c3;
16                     if a.shape[1]<=c3: c3=a.shape[1]-(c3-a.shape[1]+2);
17                     soma = soma+a[l3,c3]
18             b[l,c]=round(soma/9)
19     return b
20
21 a=cv2.imread("lion.png",0)
22 b=mediamov(a)
23 cv2.imwrite("media-py.png",b)
```

Programa 3: Filtro média móvel 3×3 em Python.

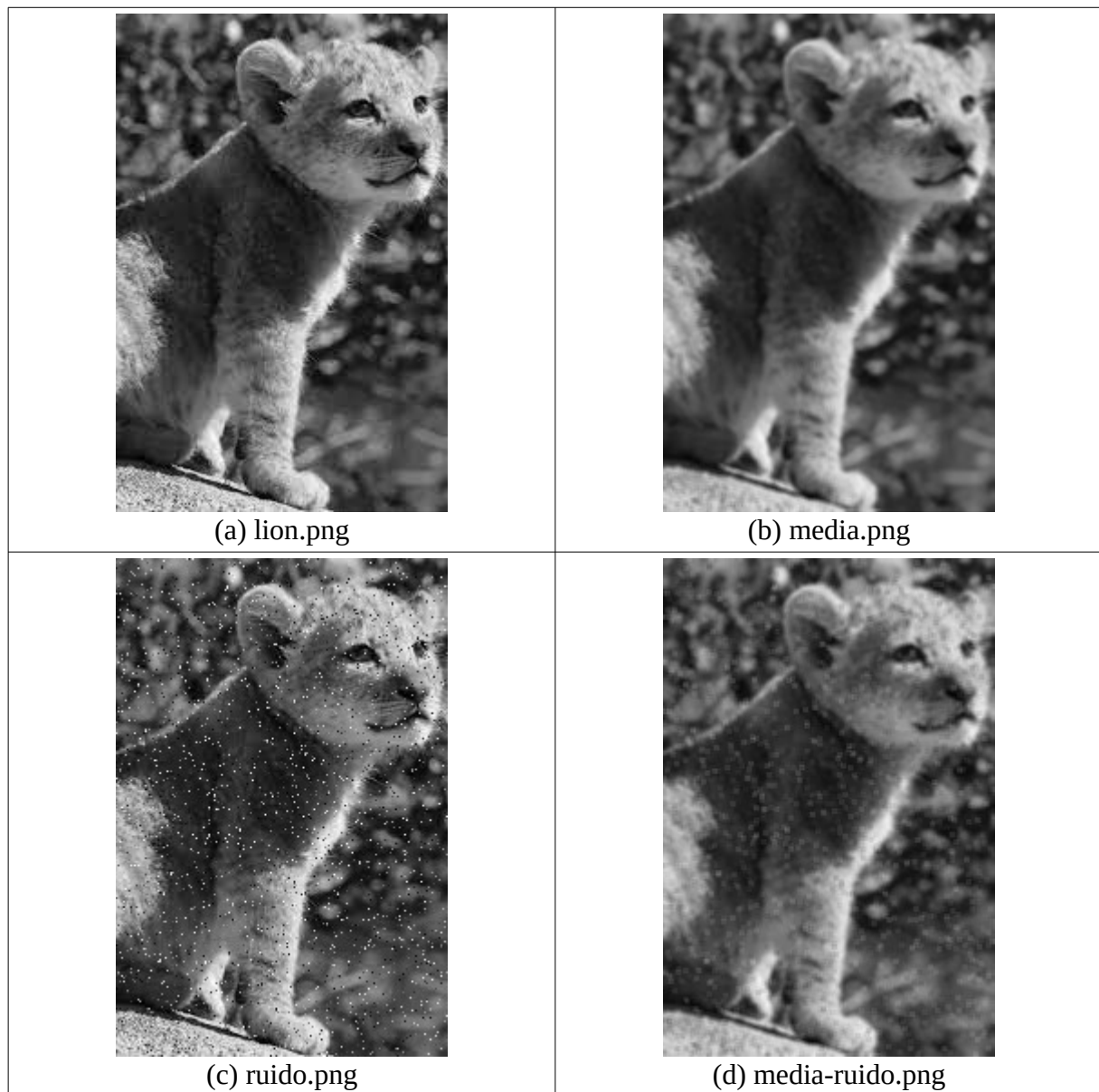


Figura 4: Exemplos de imagens com respectivas saídas geradas pelo filtro média móvel 3×3 .

Nos programas 1-3, escrevemos manualmente o filtro média móvel. Mas isto não é necessário, pois OpenCV possui uma implementação de média móvel pronta para ser usada (tanto em C++ como em Python):

C++: `void blur(const Mat& src, Mat& dst,
 Size ksize, Point anchor=Point(-1,-1),
 int borderType=BORDER_DEFAULT);`

Exemplo em C++: `blur(ent,sai,Size(3,3));`

Python: `cv2.blur(src, ksize[, dst[, anchor[, borderType]]]) → dst`

Exemplo em Python: `sai=cv2.blur(ent,(3,3))`

Usando a função `blur` do OpenCV, obtemos o programa 4 em C++. As saídas dos programas 1 a 4 são exatamente iguais. [Ou seria que as saídas dos programas 2 e 4 são exatamente iguais?]

```
//blur.cpp - 2024
//gfedcb|abcdefgh|gfedcba
#include <opencv2/opencv.hpp>
using namespace std; using namespace cv;
int main() {
    Mat_<uchar> a,b;
    a=imread("lion.png",0);
    blur(a,b,Size(3,3));
    imwrite("blur.png",b);
}
```

Programa 4: Média móvel 3×3 usando função `blur` do OpenCV.

Nota: O manual do OpenCV está disponível na internet:

<https://docs.opencv.org/4.x/index.html>

Para quem instalou Cekeikon, manuais das versões 2 e 3 estão em:

(...)/cekeikon5/opencv2docs/opencv2refman.pdf

(...)/cekeikon5/opencv3docs/index.html

Exercício: Cronometre a velocidade de processamento de 4 programas, fornecendo como entrada uma imagem grande para que a diferença de velocidade fique mais evidente.

- Média móvel escrito manualmente em C++ (programa 2).
- Média móvel escrito manualmente em Python (programa 3).
- Média móvel escrito usando rotina `blur` do OpenCV em C++ (programa 4)
- Média móvel escrito usando rotina `blur` do OpenCV em Python.

É recomendável ignorar o tempo da primeira execução, pois o computador pode demorar para carregar o interpretador ou as bibliotecas.

Exercício: Suponha que temos uma imagem grande (digamos 2.000×2.000 pixels) e queremos aplicar uma média móvel com janela grande (digamos 100×100 pixels). Neste caso, precisamos fazer uma quantidade grande de somas por pixel (10.000 somas). Pode ser que as 3 ideias abaixo acelerem o processamento (ignore possíveis pequenas diferenças nas bordas das imagens):

- Quando a janela se move um pixel para direita, não é necessário calcular a soma de todo o conteúdo da janela novamente. Basta somar, na soma da janela na posição anterior, a soma da coluna da direita (coluna que vai entrar dentro da janela) e subtrair a soma da coluna da esquerda (coluna que vai sair da janela).
- Se você calcula média móvel 100×1 seguida pela média móvel 1×100 , o resultado fica igual à média móvel 100×100 . A vantagem é que vai efetuar 200 somas por pixel em vez de 10.000. Esta propriedade é denominada de *separabilidade*.
- É possível calcular rapidamente a somatória dentro de qualquer retângulo de uma imagem calculando a imagem integral. Veja a apostila “integral” ou [WikiIntegral].

Qual (ou quais) dessas 3 ideias realmente funciona? Se houver duas ou mais ideias que funcionarem, verifique qual delas efetua o menor número de operações.

Exercício: Aplique um filtro média móvel 21×21 na imagem Lenna de uma vez e depois usando separabilidade (isto é, aplique filtro média móvel 21×1 seguido por outro filtro média móvel 1×21). Verifique se as duas imagens resultantes são iguais ou diferentes. Verifique a diferença no tempo de execução.

3. Programa ruído

O programa abaixo muda em média o valor de um pixel em cada 20 pixels, substituindo-o por um valor aleatório entre 0 e 255. A sua saída está nas figuras 4c e 5a. Estou deixando este programa explícito, somente para que vocês saibam que tipo de ruído foi inserido.

```
1 //ruído.cpp - pos2016
2 #include <opencv2/opencv.hpp>
3 using namespace std; using namespace cv;
4 Mat_<uchar> ruído(Mat_<uchar> a) {
5     Mat_<uchar> b=a.clone();
6     srand(7);
7     for (unsigned l=0; l<a.rows; l++) {
8         for (unsigned c=0; c<a.cols; c++) {
9             if (rand()%20==0) {
10                 b(l,c)=rand()%256;
11             }
12         }
13     }
14     return b;
15 }
16
17 int main() {
18     Mat_<uchar> a=imread("lion.png",0);
19     Mat_<uchar> b=ruído(a);
20     imwrite("ruído.png",b);
21 }
```

Programa 5: Programa ruído usado para gerar imagem Lenna ruidosa.

4. Filtro mediana

O segundo filtro que vamos estudar é o filtro mediana. Este filtro pertence à classe de filtros de estatística de ordem. Ele calcula mediana em vez da média aritmética. A mediana do vetor $\{a_1, \dots, a_n\}$ é o elemento que ficaria no meio do vetor se fosse ordenado (para n ímpar). Se n for par, calcula-se a média aritmética entre os dois elementos centrais do vetor ordenado.

O programa abaixo utiliza a função *nth_element*, da biblioteca-padrão de C++, para calcular o filtro mediana em janela 3×3. A função *nth_element* devolve o elemento que ficaria na n -ésima posição se o vetor fosse ordenado em ordem crescente. Note que, como a janela é fixa em 3×3, não há preocupação de tratar o caso de comprimento do vetor ser par.

```
1 //mediana.cpp - 2024
2 #include <opencv2/opencv.hpp>
3 using namespace std;
4 using namespace cv;
5
6 Mat_<uchar> mediana(Mat_<uchar> a) {
7     Mat_<uchar> b(a.rows,a.cols);
8     vector<int> v;
9     for (int l=0; l<b.rows; l++)
10         for (int c=0; c<b.cols; c++) {
11             v.resize(0);
12             for (int l2=-1; l2<=1; l2++)
13                 for (int c2=-1; c2<=1; c2++) {
14                     int l3=l+l2; int c3=c+c2;
15                     if (l3<0) l3=-l3;
16                     if (a.rows<=l3) l3=a.rows-(l3-a.rows+2);
17                     if (c3<0) c3=-c3;
18                     if (a.cols<=c3) c3=a.cols-(c3-a.cols+2);
19                     v.push_back(a(l3,c3));
20                 }
21             //vector<int>::iterator meio=v.begin()+v.size()/2; // Ou
22             auto meio=v.begin()+v.size()/2;
23             nth_element(v.begin(), meio, v.end());
24             b(l,c) = *meio;
25         }
26     return b;
27 }
28
29 int main() {
30     Mat_<uchar> a=imread("ruído.png",0);
31     Mat_<uchar> b=mediana(a);
32     imwrite("mediana_ruído.png",b);
33 }
```

Programa 6: Filtro mediana 3×3 usando função *nth_element* da biblioteca padrão de C++.

Na linha 19, o programa copia os elementos dentro da janela para o vetor *v*, que cresce dinamicamente com o método *v.push_back*. A linha 22 calcula o “iterator” para a posição central do vetor.

A linha 23 faz ordenação parcial do vetor usando a função *nth_element*. Esta função ordena somente até que a mediana de *v* fique na posição central do vetor *v*. A linha 24 pega o elemento apontado pelo iterator *meio* (isto é, a mediana do vetor) e armazena-o na imagem de saída.

Exercício: Traduza o programa 6 (mediana.cpp) para Python.

Nota: Apostilas antigas trazem a implementação da função *nth_element* “from scratch”.

Aqui, também existe uma função pronta do OpenCV que calcula filtro mediana. Aliás, a implementação do OpenCV é altamente otimizada, bem mais rápida do que a nossa implementação manual.

Nota: O artigo [Perreault2007] descreve um algoritmo muito rápido para calcular filtro mediana (não sei se é exatamente este algoritmo que está implementado dentro do OpenCV).

C++: `void medianBlur(const Mat& src, Mat& dst, int ksize);`

ksize é o tamanho da janela e tem que ser ímpar.

Ex: `medianBlur(ent, sai, 3);`

Python: `cv2.medianBlur(src, ksize[, dst]) → dst`

ksize é o tamanho da janela e tem que ser ímpar.

Ex: `sai=cv2.medianBlur(ent,3)`

A figura 5 mostra a aplicação dos filtros média móvel e mediana na imagem *ruído.png*. O filtro mediana gera uma saída “quase milagrosa”: o ruído simplesmente desapareceu!

Exercício: Dê uma explicação de por que o filtro mediana consegue fazer desaparecer quase completamente o ruído, enquanto que o filtro média móvel só “espalha” o ruído.

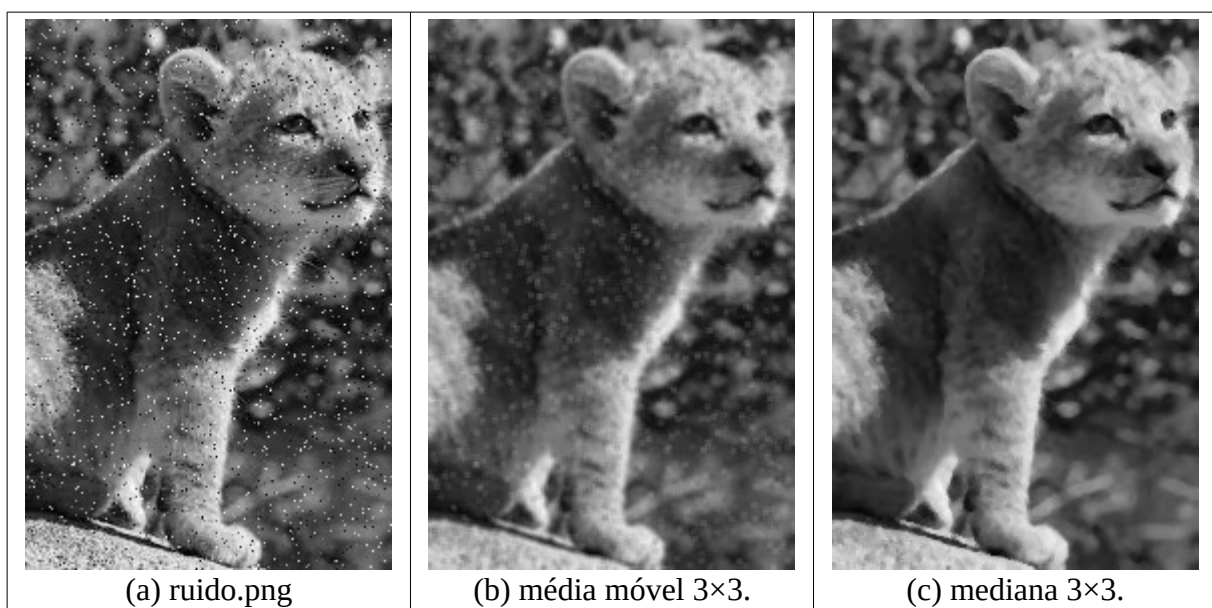


Figura 5: Aplicação do filtro média móvel 3×3 e mediana 3×3 na imagem ruído.png.

Exercício: Pense num outro filtro, diferente do filtro mediana, que também conseguiria eliminar o ruído da imagem ruído.png.



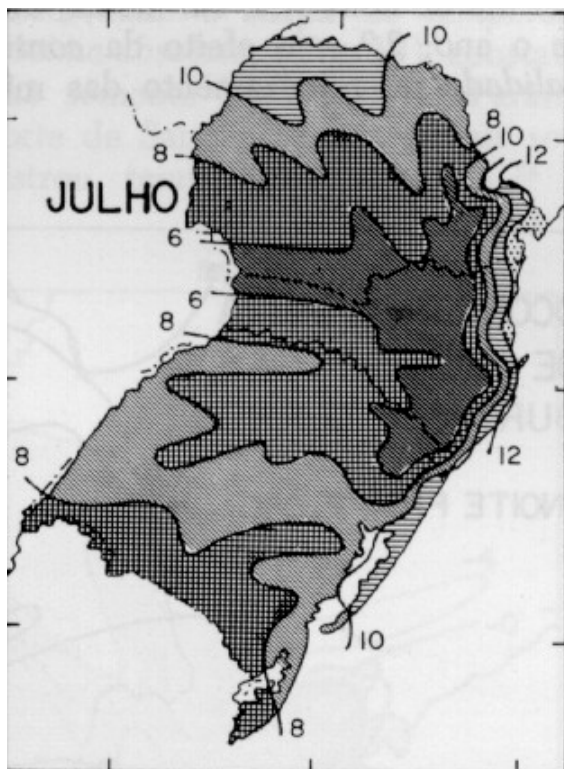
Imagem ruidosa b-rui.bmp



Imagem filtrada pela mediana

Figura 6: Exemplo de atenuação de ruído de uma imagem binária usando filtro mediana.

A figura 6 mostra um exemplo atenuação de ruído numa imagem binária pelo filtro mediana.
A figura 7 mostra uma outra aplicação de filtro mediana.



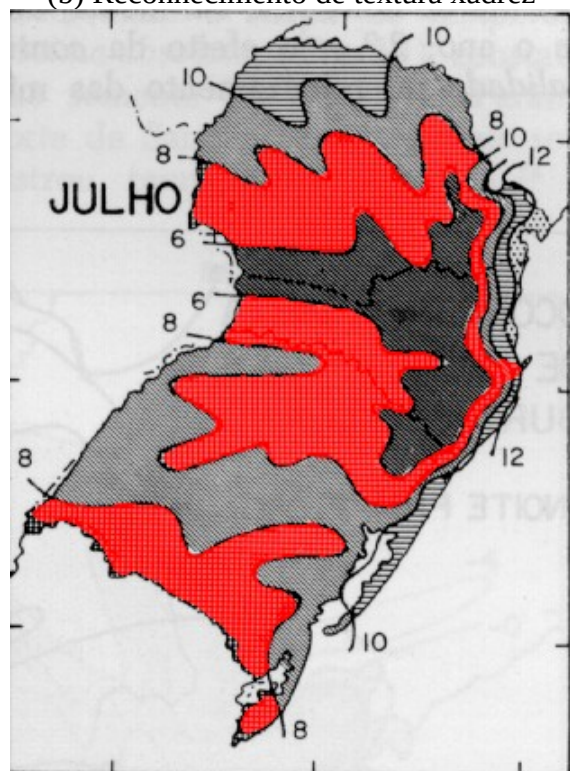
(a) Imagem original



(b) Reconhecimento de textura xadrez



(c) Filtrado pela mediana



(d) Imagem sobreposta

Figura 7: Um certo filtro conseguiu reconhecer a textura xadrez da imagem original (a) gerando a imagem com textura xadrez (b) mas está muito ruidosa. Aplicando repetidamente filtro mediana, conseguiu-se eliminar o ruído (c). Sobrepondo à imagem original, vemos que o reconhecimento de textura xadrez foi bastante satisfatório.

Exercício: Explique como é possível acelerar o cálculo de filtro mediana em imagens binárias contando o número de pixels brancos e pretos dentro da janela. Explique como é possível diminuir ainda mais o número de operações aproveitando que se conhece o número de pixels brancos e pretos da janela anterior (à esquerda).

[PSI5790-2025 aula 2. Lição de casa #1 (de 2). Vale 5 pontos.] Escreva um programa que usa o filtro mediana (usando a função `medianBlur` do OpenCV ou o filtro implementado “manualmente”) para filtrar a imagem ruidosa `fever-1.pgm` e `fever-2.pgm` (que se encontram dentro do arquivo “`filtlin.zip`” diretório “`textura`”) obtendo as imagens limpas. Figura 8 mostra a saída esperada filtrando `fever-2.pgm`.

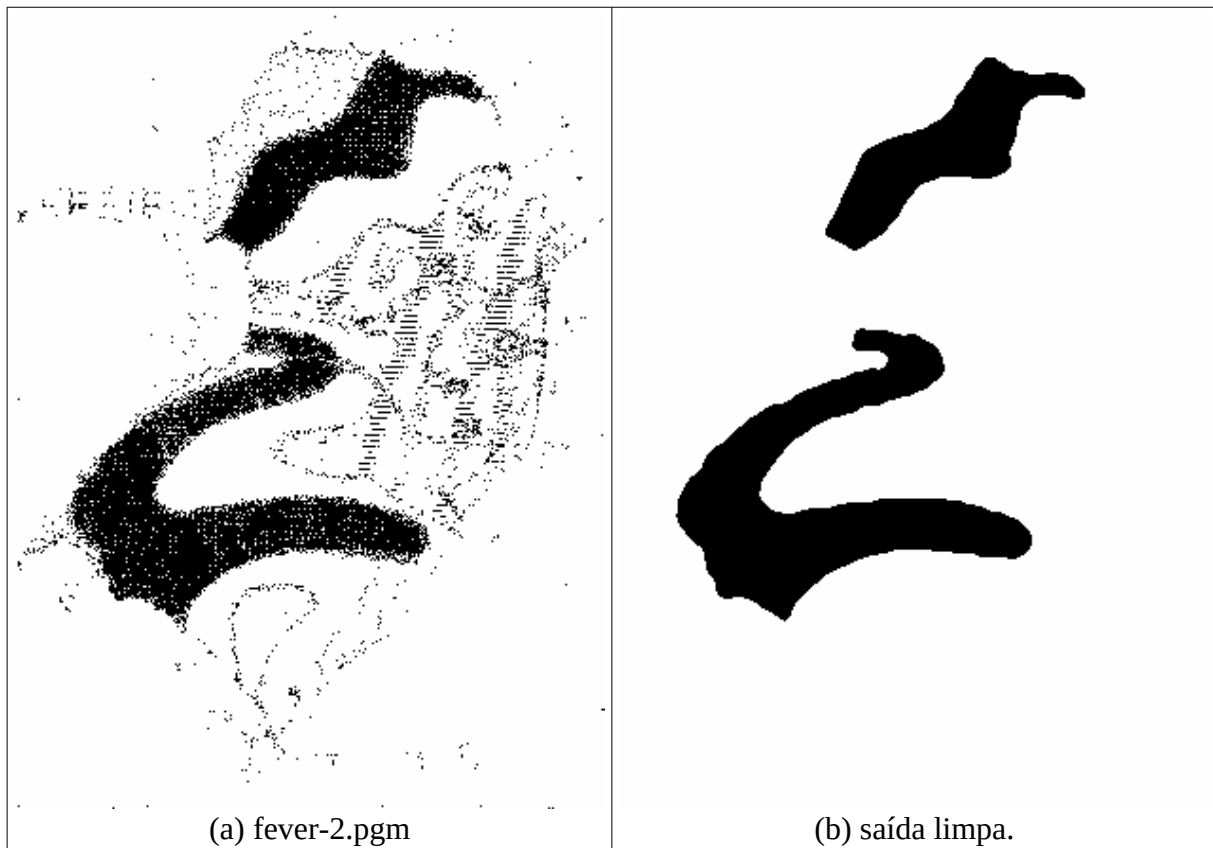


Figura 8: Uso de filtro mediana para limpar a imagem.

Exercício: Pense alguma outra aplicação onde o filtro mediana seria útil.

5. Filtros lineares espaciais (convoluções)

Definição: Um *filtro linear espacial* calcula a média aritmética ponderada dos pixels da janela. Os pesos são definidos através de matriz denominada de filtro, operador, máscara, núcleo, modelo, pesos ou janela (filter, operator, mask, kernel, template, weights or window). A filtração linear de uma imagem f de tamanho $M \times N$ por um núcleo w de tamanho $m \times n$ resultando na imagem g é dada pela expressão [Gonzalez2002]:

$$g(x, y) = \sum_{s=-a}^a \sum_{t=-b}^b w(s, t) f(x+s, y+t)$$

onde $a = (m-1)/2$ e $b = (n-1)/2$.

Nota: Na teoria, convolução é igual ao filtro linear com peso rotacionado de 180° . Na prática, em Visão Computacional, os termos filtro linear e convolução costumam ser utilizados como sinônimos.

A saída de um filtro linear depende também de dois modos de operação:

- 1) Tamanho da imagem de saída (“modo” que Matlab chama de “valid”, “same” ou “full”).
- 2) O que fazer quando se acessa um pixel fora do domínio na imagem de entrada (“tipo de borda”).

Exemplo: Vamos filtrar a imagem f com peso w , obtendo a imagem g .

$$w = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \text{ e } f = \begin{bmatrix} 2 & 2 & 2 \\ 3 & 3 & 3 \\ 4 & 4 & 4 \end{bmatrix}$$

Vamos considerar que fora do domínio da imagem f tem valor constante igual a zero. Neste caso, as saídas são:

Modo “valid” (somente os pixels onde w cabe inteiramente dentro de f são armazenados na saída g):

27

Modo “same”, atribuindo valor zero para pixels fora do domínio de f (a saída g tem o mesmo tamanho que f):

10	15	10
18	27	18
14	21	14

Nota: Não vou explicar modo “full”, pois não é muito usado.

A função de OpenCV que implementa filtro linear é *filter2D*. Esta função trabalha sempre no modo “same”.

Nota: Veremos mais adiante que a função *matchTemplate*, que também implementa filtro linear, trabalha sempre no modo “valid”.

Para não nos preocuparmos com erros de “overflow”, números negativos e arredondamentos, vamos usar *filter2D* sempre com todas as matrizes envolvidas (imagem de entrada, peso e imagem de saída) do tipo `Mat_<float>`.

C++: `void filter2D(InputArray src, OutputArray dst, int ddepth, InputArray kernel, Point anchor=Point(-1,-1), double delta=0, int borderType=BORDER_DEFAULT)`

Exemplo: `filter2D(ent, sai, -1, ker);`

Nota: `ddepth = -1` indica que a matriz de saída será do mesmo tipo que a de entrada.

Python: `cv2.filter2D(src, ddepth, kernel[, dst[, anchor[, delta[, borderType]]]]) → dst`

Exemplo: `sai=cv2.filter2D(ent, -1, ker)`

A saída do filtro varia de acordo com o que fizer com os pixels fora do domínio da imagem de entrada (“borderType”). Segundo manual do OpenCV v3, as formas de tratar “borda” são:

http://docs.opencv.org/trunk/d2/de8/group_core_array.html#ga209f2f4869e304c82d07739337eae7c5

<code>BORDER_CONSTANT</code>	<code>iiiiii abcdefgh iiiiiii</code> with some specified i
<code>BORDER_REPLICATE</code>	<code>aaaaaa abcdefgh hhhhhhh</code>
<code>BORDER_REFLECT</code>	<code>fedcba abcdefgh hgfedcb</code>
<code>BORDER_WRAP</code>	<code>cdefgh abcdefgh abcdefg</code>
<code>BORDER_REFLECT_101</code>	<code>gfedcb abcdefgh gfedcba</code>
<code>BORDER_DEFAULT</code>	same as <code>BORDER_REFLECT_101</code>

Sendo que o default é `BORDER_DEFAULT` ou `BORDER_REFLECT_101`.

Alguns exemplos de como OpenCV enxerga os pixels fora do domínio de uma matriz 3×3:

$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ <p>(a) Matriz 3×3.</p>	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ 0 & 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ <p>(b) <code>BORDER_CONSTANT</code> com background 0</p>
$\begin{bmatrix} 1 & 1 & 2 & 3 & 3 \\ 1 & 1 & 2 & 3 & 3 \\ 4 & 4 & 5 & 6 & 5 \\ 7 & 7 & 8 & 9 & 9 \\ 7 & 7 & 8 & 9 & 9 \end{bmatrix}$ <p>(c) <code>BORDER_REPLICATE</code></p>	$\begin{bmatrix} 5 & 4 & 5 & 6 & 5 \\ 2 & 1 & 2 & 3 & 2 \\ 5 & 4 & 5 & 6 & 5 \\ 8 & 7 & 8 & 9 & 8 \\ 5 & 4 & 5 & 6 & 5 \end{bmatrix}$ <p>(d) <code>BORDER_DEFAULT</code></p>

Figura 9: Como OpenCV enxerga os pixels fora do domínio de acordo com “modo”.

Há algumas funções que gostaríamos que existisse em C++/OpenCV. Para não ter que escrever esse código em todos os programas, vamos escrevê-los num arquivo chamado “[procimagem.h](#)”, deixá-lo no mesmo diretório dos programas, e incluí-lo no início dos programas. Essas funções estão incluídos dentro da biblioteca Cekeikon mas estamos escrevendo-as explicitamente aqui para que não precise usar essa biblioteca e para que fique mais fácil entender o que está acontecendo. Por enquanto, esse arquivo fica:

```

1 #include <opencv2/opencv.hpp>
2 #include <iostream>
3 using namespace std;
4 using namespace cv;
5
6 void erro(string s1="") {
7     cerr << s1 << endl;
8     exit(1);
9 }
10
11 Mat_<float> filtro2d(Mat_<float> ent, Mat_<float> ker, int borderType=BORDER_DEFAULT)
12 { Mat_<float> sai;
13   filter2D(ent,sai,-1,ker,Point(-1,-1),0.0,borderType);
14   return sai;
15 }
16
17 Mat_<Vec3f> filtro2d(Mat_<Vec3f> ent, Mat_<float> ker, int borderType=BORDER_DEFAULT)
18 { Mat_<Vec3f> sai;
19   filter2D(ent,sai,-1,ker,Point(-1,-1),0.0,borderType);
20   return sai;
21 }

```

Programa 7: Arquivo *procimagem.h* desta aula.

Esse arquivo tem, por enquanto, duas funções:

- 1) *erro*: Imprime mensagem de erro e aborta o programa.
- 2) *filtro2d*: Facilita chamar a função filter2D sem precisar passar monte de parâmetros.

Assim, a chamada (1) com essa função equivale à chamada (2) em OpenCV puro:

- 1) Mat_<float> sai = filtro2d(ent,ker);
- 2a) Mat_<float> sai; filter2D(ent,sai,-1,ker,Point(-1,-1),0,BORDER_DEFAULT); **OU**
- 2b) Mat_<float> sai; filter2D(ent,sai,-1,ker);

- 1) Mat_<float> sai = filtro2d(ent,ker,BORDER_REPLICATE);
- 2) Mat_<float> sai; filter2D(ent,sai,-1,ker,Point(-1,-1),0,BORDER_REPLICATE);

- 1) Mat_<float> sai = filtro2d(ent,ker,BORDER_CONSTANT);
- 2) Mat_<float> sai; filter2D(ent,sai,-1,ker,Point(-1,-1),0,BORDER_CONSTANT);

Nota: O site abaixo explica como compilar programas C++/OpenCV ou C++/OpenCV/Cekeikon:
<http://www.lps.usp.br/hae/software>

6. Exemplos de filtros lineares

A figura 10 mostra alguns exemplos de filtros lineares.

$\frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ <p>(a) Filtro espacial passa-baixa (média móvel)</p>	
$\frac{1}{8} \times \begin{bmatrix} -1 & -1 & -1 \\ -1 & +8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$ <p>(b1) Filtro espacial passa-alta. [Laplace ou 2ª derivada]</p>	$\frac{1}{4} \times \begin{bmatrix} 0 & -1 & 0 \\ -1 & +4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ <p>(b2) Filtro espacial passa-alta alternativa. [Laplace ou 2ª derivada]</p>
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ <p>(c) Impulso de Dirac</p>	$\frac{1}{10} \times \begin{bmatrix} -1 & -1 & -1 \\ -1 & 18 & -1 \\ -1 & -1 & -1 \end{bmatrix}$ <p>(d) Enfatiza arestas (Dirac+0,8×passa-alta [b1]).</p> <p>1,8 = 1 + 0,8 -0,1 = 0 + (8/10) * (-1/8)</p>

Figura 10: Exemplos de filtros lineares.

6.1 Média móvel

O filtro da figura 10a é a média móvel que já estudamos no começo desta aula. Vamos reescrever esse filtro, agora usando a função `filtro2d`. Vamos trabalhar com todas as matrizes do tipo ponto flutuante 32 bits. Você precisa deixar o arquivo “*procimagem.h*” no mesmo diretório do programa .cpp.

Nota: Em C/C++, o comando:

```
#include <nomearq.h>
```

irá incluir o arquivo *nomearq.h* que se encontra em algum dos diretórios que contém arquivos de cabeçalho do compilador. Enquanto isso:

```
#include "nomearq.h"
```

irá incluir o arquivo *nomearq.h* que se encontra no diretório corrente.

```
1 //media-movel.cpp - 2024
2 #include "procimagem.h"
3 int main(int argc, char** argv) {
4     if (argc!=3) erro("media-movel ent.pgm sai.pgm");
5     Mat_<float> ent=imread(argv[1],0);
6     if (ent.total()==0) erro("Erro leitura");
7     Mat_<float> ker= (Mat_<float>(3,3) <<
8         +1, +1, +1,
9         +1, +1, +1,
10        +1, +1, +1);
11     ker = (1.0/9.0) * ker;
12     Mat_<float> sai=filtro2d(ent,ker);
13     imwrite(argv[2],sai);
14 }
```

Programa 8: Média móvel usando função *filtro2d*.

Programa 8 lê os argumentos do sistema operacional (linha 4). Isto é, o programa acima deve ser chamado com 3 argumentos:

```
diretorio> media-movel ruido.png saida.png
```

Os 3 argumentos são:

argumento #0 (argv[0]): o nome do programa (media-movel);

argumento #1 (argv[1]): o nome da imagem de entrada (ruido.png); e

argumento #2 (argv[2]): o nome da imagem de saída (saida.png).

Na linha 5, o programa lê a imagem de entrada (cujo nome está no argv[1]) na matriz *ent*. Essa matriz tem valores que vão de 0,0 (preto) a 255,0 (branco).

Nas linhas 6-10 o programa constrói núcleo 3×3 *ker* com todos os elementos valendo 1/9. Filtrar com este núcleo equivale a calcular média móvel.

Na linha 11, o programa aplica o filtro com pesos especificados em *ker* na imagem de entrada *ent* e armazena a saída na matriz *sai*.

A linha 12 imprime a matriz de ponto flutuante *sai* como imagem, considerando que preto é 0 e branco é 255. A saída obtida é idêntica àquela mostrada na figura 4b.

O mesmo programa está escrito abaixo em Python.

```
1 #media-movel.py - 2024
2 import cv2
3 import sys
4 import numpy as np
5
6 if len(sys.argv)!=3:
7     sys.exit("media-movel ent.pgm sai.pgm")
8
9 ent=cv2.imread(sys.argv[1],0)
10
11 ker=[[1,1,1], [1,1,1], [1,1,1]]
12 ker=np.float32(ker)
13 ker=(1.0/9.0)*ker
14
15 sai=cv2.filter2D(ent,-1,ker)
16 cv2.imwrite(sys.argv[2],sai)
```

Programa 9: Média móvel usando função *filter2D* em Python.

6.2 Filtro passa-alta [Laplace ou 2ª derivada]

Modificando os pesos no programa 8, obtemos o filtro passa-alta (programa 10), também chamado de Laplace. Esse filtro detecta a 2ª derivada da imagem. Fazendo analogia com eletricidade, filtro passa-alta funciona como um capacitor que bloqueia o componente DC do sinal, deixando passar somente o componente AC.

O filtro passa-alta elimina regiões da imagem com nível de cinza constante. Essas regiões passam a ter valor zero após a filtragem. Este filtro detecta as mudanças de nível de cinza, isto é, passagem de uma região escura para outra clara ou de uma região clara para escura.

A matriz resultante da filtragem vai oscilar em torno de zero. Se imprimíssemos essa matriz diretamente, todos os pixels negativos (que são aproximadamente a metade da imagem) seriam impressos como pretos, pois a função *imwrite* considera preto qualquer valor menor ou igual a zero.

Para evitar isso, na linha 13, o programa soma 128 a todos os pixels da imagem. Isso faz com que os pixels negativos da matriz sejam visualizados como cinza escura e pixels positivos sejam visualizados como cinza clara. Além disso, o programa multiplica os valores dos pixels por 5 (antes de somar 128) para ampliar as flutuações em torno do zero. A saída desse programa está na figura 11.

```
1 //highpass.cpp - 2024
2 #include "procimagem.h"
3 int main(int argc, char** argv) {
4     if (argc!=3) erro("highpass ent.pgm sai.pgm");
5     Mat_<float> ent=imread(argv[1],0);
6     if (ent.total()==0) erro("Erro leitura");
7     Mat_<float> ker= (Mat_<float>(3,3) <<
8         -1, -1, -1,
9         -1, +8, -1,
10        -1, -1, -1);
11     ker = (1.0/9.0) * ker;
12     Mat_<float> sai = filtro2d(ent,ker);
13     sai = 128 + 5 * sai;
14     imwrite(argv[2],sai);
15 }
```

Programa 10: Filtro passa-alta.

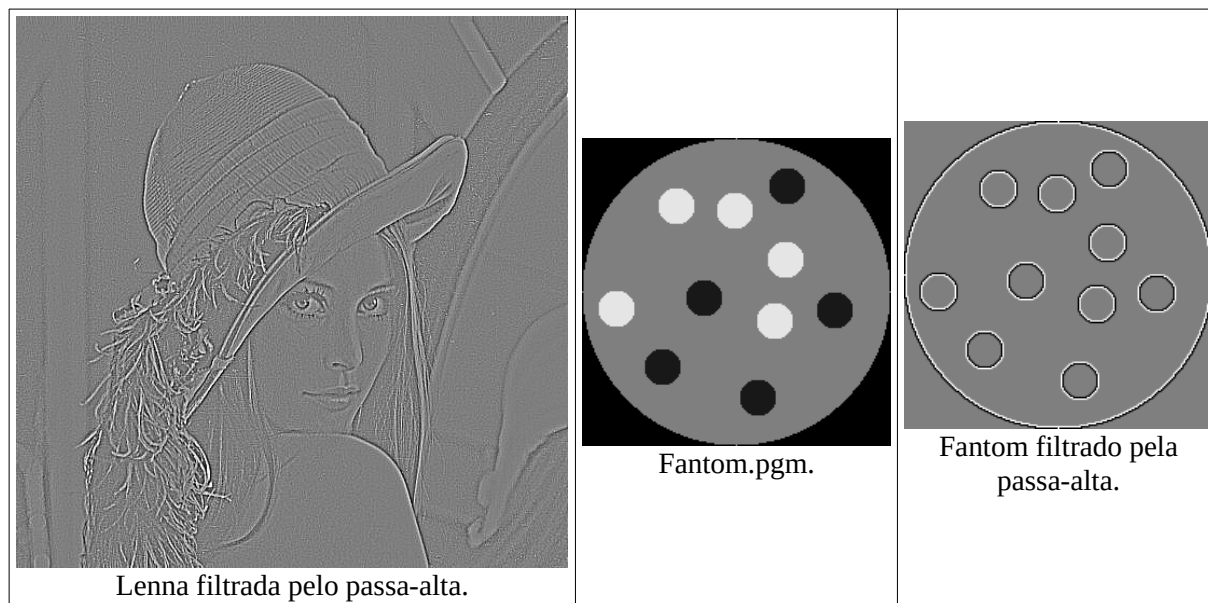


Figura 11: Saída do filtro passa-alta (Laplace ou 2ª derivada). Cinza escura representa pixels negativos. Cinza clara representa pixels positivos.

6.3 Impulso de Dirac

A figura 10c mostra o núcleo do impulso de Dirac. É fácil constatar que, aplicando esse filtro, a imagem de saída será exatamente igual à imagem de entrada.

6.4 Filtro enfatiza bordas

O filtro “enfatiza bordas” é uma combinação linear do impulso de Dirac com filtro passa-alta: $\text{Dirac} + \text{peso} \times \text{passa-alta}$. Este filtro reforça as altas frequências, fazendo a saída parecer “mais nítida” do que a entrada. Fazendo analogia com sinais, seria como um equalizador que amplifica altas frequências (figuras 12 e 13). É possível reforçar mais ou menos as altas frequências aumentando ou diminuindo o peso 0,8.

Figuras 13a-b aplica esse filtro numa imagem sintética, o que permite perceber mais claramente o que está acontecendo. Repare que em toda transição de região escura para região clara, o filtro cria uma fina borda (ou contorno) com cinza mais escura seguida por outra borda com cinza mais clara. E contrário na transição de claro para escuro. Isto faz imagem parecer mais nítida.

```

1 //enfboarda.cpp - 2024
2 #include "procimagem.h"
3 int main(int argc, char** argv) {
4     if (argc!=4) erro("enfboarda ent.pgm sai.pgm peso");
5     Mat_<float> ent=imread(argv[1],0);
6     if (ent.total()==0) erro("Erro: Leitura");
7
8     Mat_<float> dirac= (Mat_<float>(3,3) <<
9         0, 0, 0,
10        0, 1, 0,
11        0, 0, 0);
12
13     Mat_<float> highpass= (Mat_<float>(3,3) <<
14         -1, -1, -1,
15         -1, +8, -1,
16         -1, -1, -1);
17     highpass = (1.0/8.0)*highpass;
18
19     double peso;
20     sscanf(argv[3], "%lf", &peso);
21     Mat_<float> enfboarda = dirac + peso * highpass;
22
23     Mat_<float> sai = filtro2d(ent, enfboarda);
24     imwrite(argv[2], sai);
25 }

```

Programa 11: Filtro “ênfatisa borda” é uma combinação de impulso de Dirac com passa alta.

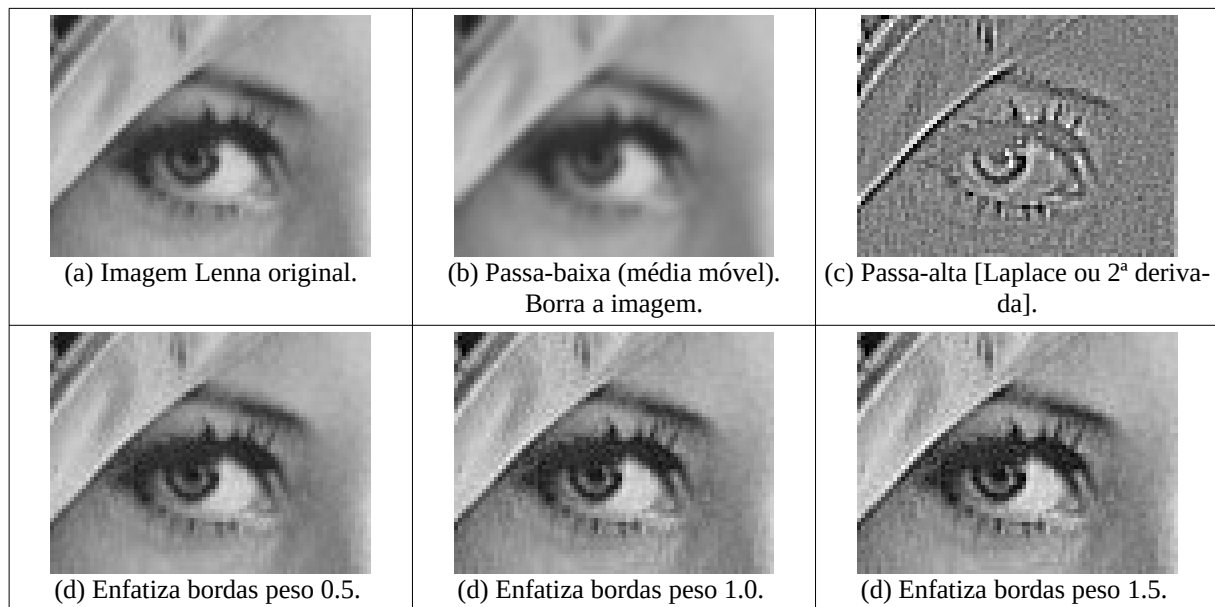


Figura 12: O filtro “ênfatisa bordas” é uma combinação linear de impulso de Dirac (isto é, a imagem original) com passa-alta. Perceptualmente, parece que a imagem fica mais nítida após ênfatisar as bordas.

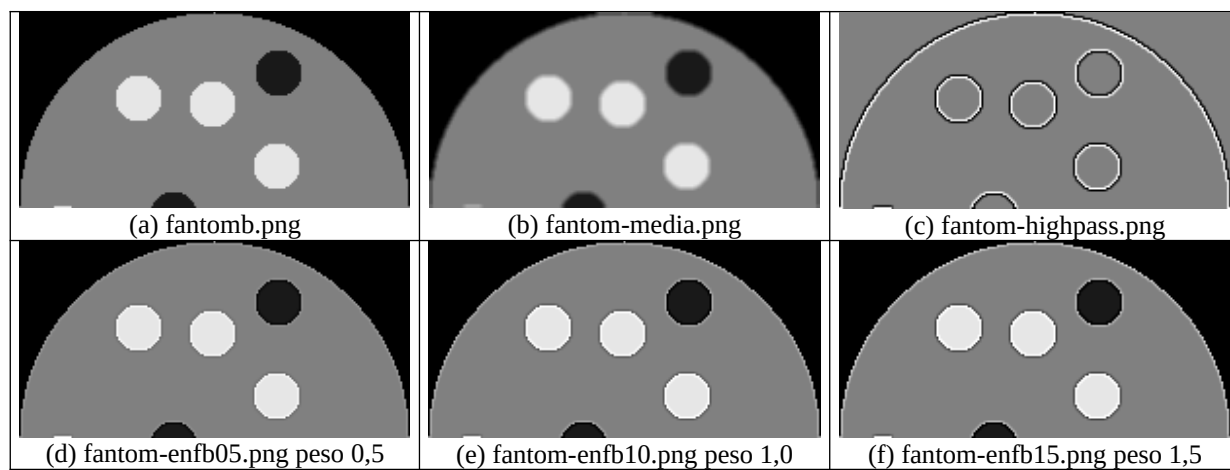


Figura 13: Coluna esquerda: imagens originais. Coluna direita: Saída do filtro “ênfatiza bordas”.

Exercício: Experimente aplicar o filtro “ênfatiza bordas” em diferentes imagens.

6.5 Aplicar um filtro nas 3 bandas da imagem colorida

O programa abaixo aplica o filtro enfatiza bordas nas três bandas de uma imagem colorida, com peso 0.8. Vec3f significa vetor de 3 variáveis float.

```
//enfboardac.cpp - 2024
#include "procimagem.h"

int main(int argc, char** argv) {
    if (argc!=3) erro("enfboardac ent.ppm sai.ppm");
    Mat_<Vec3f> ent=imread(argv[1],1);
    if (ent.total()==0) erro("Erro leitura");
    Mat_<float> ker= (Mat_<float>{3,3} <<
        -1, -1, -1,
        -1, 18, -1,
        -1, -1, -1);
    ker = (1.0/10.0) * ker;
    Mat_<Vec3f> sai = filtro2d(ent,ker);
    imwrite(argv[2],sai);
}
```



casa.ppm



casa.ppm com ênfatização de bordas.

Figura F2: Enfatizando as bordas de uma imagem colorida, obtemos uma imagem que parece mais nítida que a original.

6.5 Filtro Gaussiano

Um filtro gaussiano (Gaussian blur ou Gaussian smoothing) é o filtro linear que borra uma imagem usando núcleo gaussiano. É amplamente usado para reduzir ruído da imagem, para reduzir detalhes e também como pré-processamento para obter imagens em diferentes escalas (espaço de escala).

A função gaussiana 1-D de média zero e desvio-padrão σ é:

$$g_{\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{x^2}{2\sigma^2}\right]$$

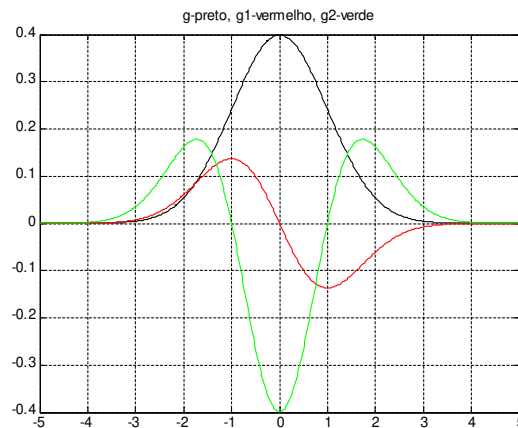


Figura 14: Função gaussiana 1-D com $\mu=0$ e $\sigma=1$ (preto); 1ª derivada (vermelho); e 2ª derivada (verde).

A função gaussiana 2-D de média zero e desvio-padrão σ é:

$$g_{\sigma}(x, y) = \frac{1}{2\pi\sigma^2} \exp\left[-\frac{x^2 + y^2}{2\sigma^2}\right]$$

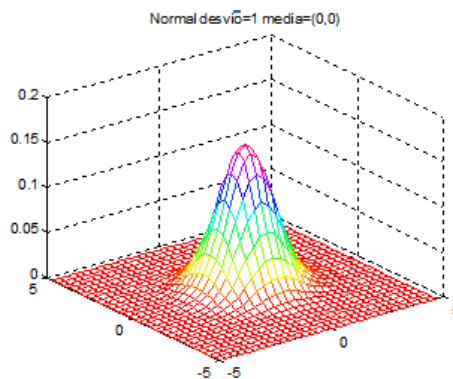


Figura 15: Função gaussiana 2-D com $\mu=(0,0)$ e $\sigma=1$.

O filtro gaussiano é separável, isto é, aplicar dois filtros gaussianos 1-D nas duas direções, um após o outro (horizontal seguido por vertical ou vice-versa), equivale a aplicar o filtro 2-D. Esta propriedade torna o filtro gaussiano computacionalmente muito eficiente.

Figura 16 mostra a imagem “casa.pgm” filtrada com núcleo gaussiano de diferentes desvios-padrões. Note como a imagem vai ficando borrada.

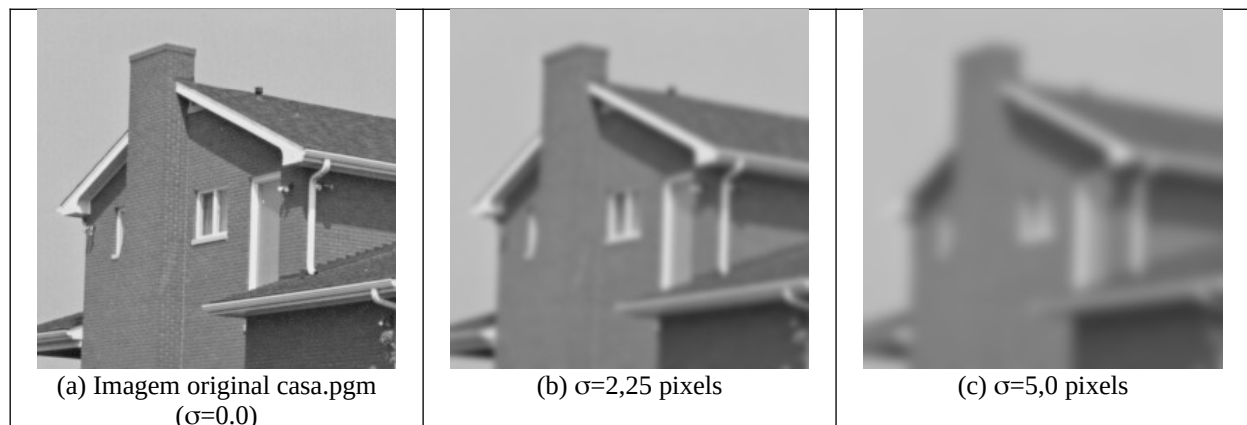


Figura 16: Imagem filtrada com filtro gaussiano com diferentes escalas.

Existe uma função pronta (GaussianBlur) em OpenCV que aplica o filtro gaussiano. Os programas abaixo aplicam o filtro gaussiano com desvio-padrão de 5 pixels.

“Size(0,0)” indica que a função deve calcular o tamanho conveniente de núcleo a partir do desvio-padrão. Veja o manual do OpenCV para maiores detalhes.

A função *imshow* para imagens *float* assume que 0 é preto e 1 é branco. Assim, é necessário dividir os pixels por 255,0 antes de chamá-la.

```
//gaussian.cpp 2024
#include "procimagem.h"
int main() {
    Mat_<float> a=imread("casa.pgm",0);
    if (a.total()==0) erro("Erro leitura");
    Mat_<float> b;
    GaussianBlur(a,b,Size(0,0),5);
    b=b/255.0;
    imshow("janela",b);
    waitKey(0);
}
```

Programa 12: Filtro gaussiano em C++.

```
#gaussian.py 2024
import numpy as np
import cv2
from matplotlib import pyplot as plt
ag=cv2.imread("casa.pgm",0)
a=np.float32(ag/255.0)
b=cv2.GaussianBlur(a,(0,0),5)
plt.imshow(b,cmap="gray")
plt.show()
cv2.imshow("janela",b)
cv2.waitKey()
```

Programa 13: Filtro gaussiano em Python.

Nota: Programas 12 e 13 estão um pouco diferentes.

Nota: Para olhar o núcleo gaussiano 1-D use o programa abaixo.

```
//gausskernel.cpp 2024
#include "procimagem.h"
int main() {
    int sigma=1;
    Mat_<double> k=getGaussianKernel(2*3*sigma+1,sigma);
    cout << k << endl;
}
```

Programa 14: Imprime o núcleo gaussiano.

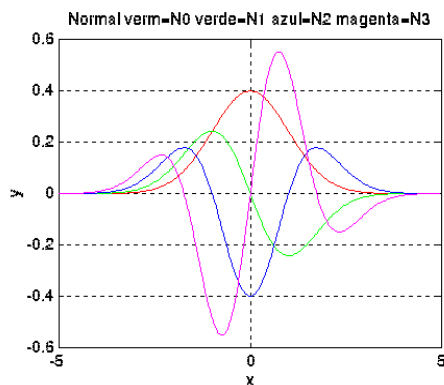
saída:

```
[0.004433048175243745;
 0.05400558262241448;
 0.2420362293761143;
 0.3990502796524549;
 0.2420362293761143;
 0.05400558262241448;
 0.004433048175243745]
```

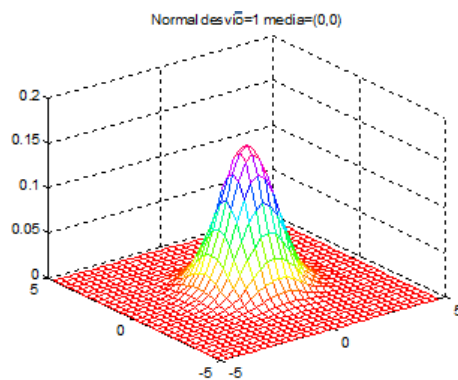
Notas:

- a) Repare na figura 14 (ou 17a) que o valor da gaussiana torna-se bem pequeno após 3σ , o que nos permite ignorar os valores do núcleo após 3σ .
- b) O núcleo gaussiano impresso é 1-D, pois o filtro usa separabilidade (filtro 1-D vertical seguido de horizontal ou vice-versa).

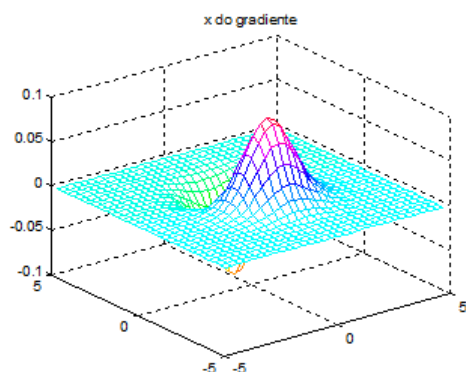
Exercício: Crie o núcleo gaussiano 2-D com desvio-padrão 1 pixel utilizando o núcleo gaussiano 1-D gerado pelo programa acima.



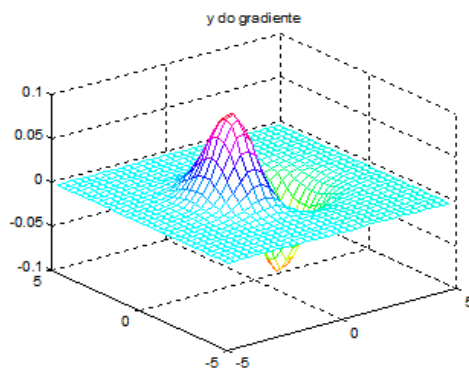
(a) Função gaussiana 1D com $\sigma=1$ (vermelho) e suas 1ª, 2ª e 3ª derivadas (respectivamente em verde, azul e magenta).



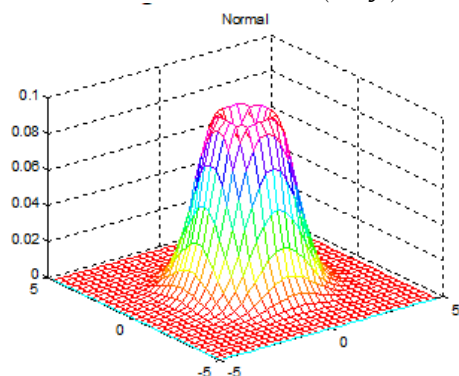
(b) Função gaussiana G (2D) com $\sigma=1$.



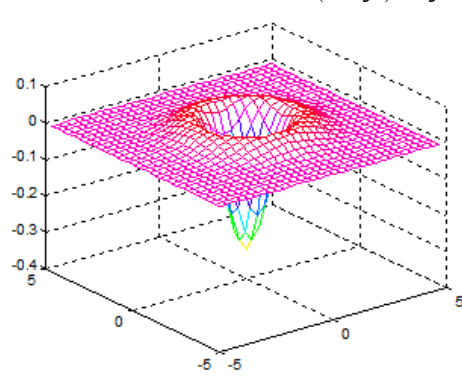
(d) Derivada parcial $x \quad \partial G(x, y) / \partial x$.



(e) Derivada parcial $y \quad \partial G(x, y) / \partial y$.



(c) Módulo do gradiente $\|\nabla G(x, y)\|$.



(f) Laplaciano da gaussiana $\nabla^2 G(x, y)$.

Figura 17: Funções gaussianas unidimensional, bidimensional e suas derivadas.

7. Gradiente

7.1 Filtros lineares para calcular gradiente

Gradiente é muito usado tanto em Processamento de Imagens como em Aprendizado de Máquina e Redes Neurais. Para se ter uma ideia intuitiva do gradiente, veja a figura 18. Gradiente é um campo vetorial (isto é, cada pixel é um vetor 2D) onde cada vetor aponta para a direção onde a imagem torna-se mais clara. Se considerar a imagem como um relevo (o nível de cinza de um pixel representa a altura do relevo naquele ponto), colocando uma bola num certo pixel, ela desceria no sentido contrário ao apontado pelo gradiente, com aceleração proporcional ao seu módulo. Como o gradiente de uma imagem em níveis de cinza é um campo vetorial, deve ser representado por 2 imagens em níveis de cinza ou como uma imagem complexa (cada pixel é um número complexo).

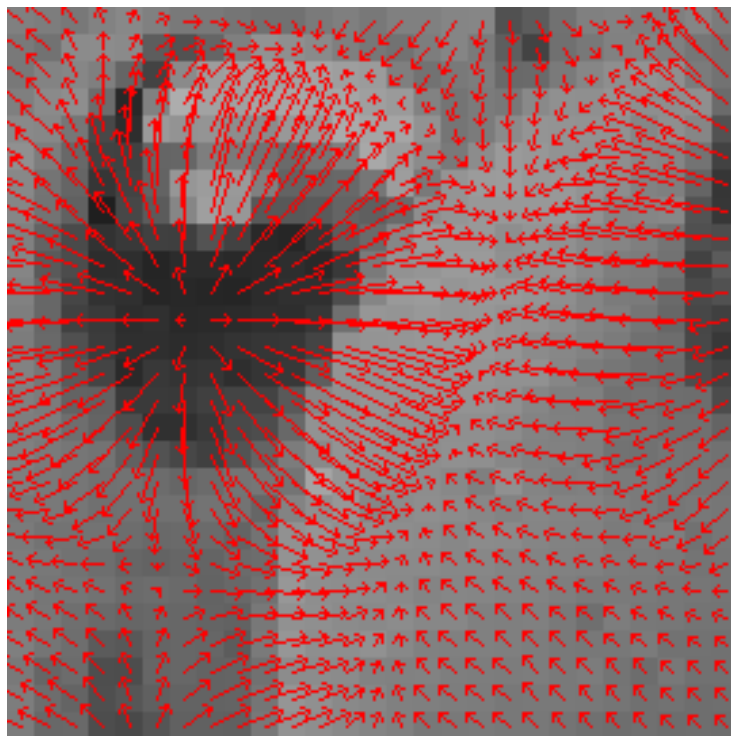


Figura 18: Gradiente desenhado em forma de flechas.

Definição: Seja f uma função $f: \mathbb{R}^2 \rightarrow \mathbb{R}$. O gradiente de f é:

$$\nabla f(x, y) = \left[\frac{\partial f(x, y)}{\partial x}, \frac{\partial f(x, y)}{\partial y} \right]$$

Isto define o gradiente de uma função contínua f . Como imagem digital é discreta tanto no domínio quanto no contradomínio, esta definição deve ser adaptada.

A derivada de um sinal digital $x[n]$ costuma ser calculada como:

$$x'(n) = \frac{x(n+1) - x(n-1)}{2}$$

Exemplo: $x[n] = [2, 3, 4, 5, 3, 1, -2]$
 $x'[n] = [?, 1, 1, -0.5, -2, -2, ?]$

Podemos aplicar a mesma ideia para calcular gradiente de uma imagem. Veja a figura 19a que mostra dois filtros lineares simples para calcular as duas derivadas parciais do gradiente. É igual à fórmula acima.

Esta ideia simples pode ser refinada sucessivamente em operadores de Prewitt, Sobel e Scharr. Uma propriedade desejável do gradiente é a simetria rotacional, isto é, gradiente de uma imagem rotacionada deve ser igual ao gradiente rotacionada da mesma imagem. Aparentemente, o gradiente calculado com núcleo de Scharr é o que melhor satisfaz essa propriedade.

Nota: A saída de um operador deve ser multiplicada por uma constante apropriada, inversamente proporcional à soma absoluta dos pesos do núcleo ($1/\text{soma_abs_pesos}$ para que inclinação 1 na imagem dê derivada 1) ou ($2/\text{soma_abs_pesos}$ para que a saída esteja no intervalo $[-\text{max}, +\text{max}]$ onde max é o maior valor possível da imagem de entrada).

$\begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ <p>(a) Filtros simples para calcular gradiente.</p>	$\frac{1}{3} \times \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$ <p>(b) Prewitt</p>
$\frac{1}{4} \times \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ <p>(c) Sobel</p>	$\frac{1}{16} \times \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix} \quad \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$ <p>(d) Scharr</p>

Figura 19: Núcleos de filtros lineares usados para calcular gradiente.

```

1 //gradiente.cpp - 2024
2 #include "procimagem.h"
3
4 void grad(Mat_<float> ent, Mat_<float>& saix, Mat_<float>& saiy)
5 { Mat_<float> mx=(Mat_<float>(3,3)<<-3.0, 0.0, +3.0,
6   -10.0, 0.0, +10.0,
7   -3.0, 0.0, +3.0);
8   mx=mx/16.0;
9   Mat_<float> my=(Mat_<float>(3,3)<<-3.0, -10.0, -3.0,
10    0.0, 0.0, 0.0,
11    +3.0, +10.0, +3.0);
12   my=my/16.0;
13   saix=filtro2d(ent,mx);
14   saiy=filtro2d(ent,my);
15 }
16
17 int main() {
18   Mat_<float> ent=imread("fantom.pgm",0);
19   if (ent.total()==0) erro("Erro leitura");
20   Mat_<float> saix;
21   Mat_<float> saiy;
22   grad(ent,saix,saiy);
23
24   Mat_<float> t;
25   t=128+saix; imwrite("gradx.png",t);
26   t=128+saiy; imwrite("grady.png",t);
27
28   Mat_<float> tx; pow(saix,2,tx);
29   Mat_<float> ty; pow(saiy,2,ty);
30   Mat_<float> modgrad; pow(tx+ty,0.5,modgrad);
31   imwrite("modgrad.png",modgrad);
32 }

```

Programa 15: Cálculo de gradiente Scharr usando *filtro2d*.

O programa 15 calcula as duas derivadas parciais de gradiente usando núcleo de Scharr. Também calcula o módulo do gradiente. As saídas deste programa estão na figura 20.

OpenCV possui os filtros de Sobel e Scharr prontos. Veja o manual para maiores detalhes. O programa 16 faz a mesma tarefa do programa15 usando a função pronta *Scharr*.

C++: `void Sobel(InputArray src, OutputArray dst, int ddepth, int dx, int dy, int ksize=3, double scale=1, double delta=0, int borderType=BORDER_DEFAULT)`

Python: `cv2.Sobel(src, ddepth, dx, dy[, dst[, ksize[, scale[, delta[, borderType]]]])` → dst

C++: `void Scharr(InputArray src, OutputArray dst, int ddepth, int dx, int dy, double scale=1, double delta=0, int borderType=BORDER_DEFAULT)`

Python: `cv2.Scharr(src, ddepth, dx, dy[, dst[, scale[, delta[, borderType]]]])` → dst

```
1 //gradiente2.cpp - 2024
2 #include "procimagem.h"
3
4 void grad(Mat_<float> ent, Mat_<float>& saix, Mat_<float>& saiy) {
5     Scharr(ent, saix, -1, 1, 0); saix=saix/16.0;
6     Scharr(ent, saiy, -1, 0, 1); saiy=saiy/16.0;
7 }
8
9 int main() {
10     Mat_<float> ent=imread("fantom.pgm",0);
11     if (ent.total()==0) erro("Erro leitura");
12     Mat_<float> saix;
13     Mat_<float> saiy;
14     grad(ent, saix, saiy);
15
16     Mat_<float> t;
17     t=128+saix; imwrite("gradx.png",t);
18     t=128+saiy; imwrite("grady.png",t);
19
20     Mat_<float> tx; pow(saix,2,tx);
21     Mat_<float> ty; pow(saiy,2,ty);
22     Mat_<float> modgrad; pow(tx+ty,0.5,modgrad);
23     imwrite("modgrad.png",modgrad);
24 }
```

Programa 16: Cálculo de gradiente usando função pronta *Scharr*.

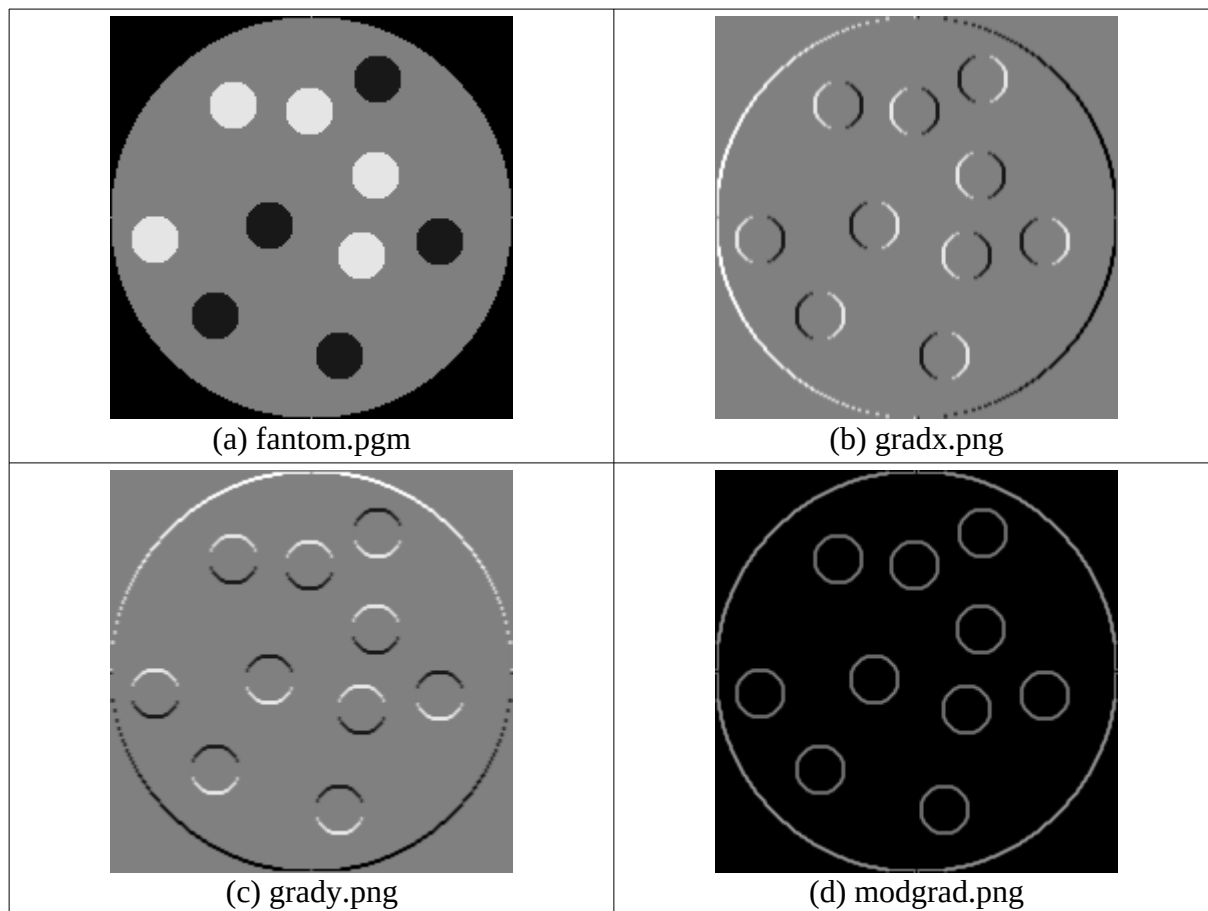


Figura 20: Gradiente calculado usando núcleo de Scharr (saída do programa15 ou 16).

Exercício: Altere o programa 15 para calcular gradiente usando núcleo de Sobel. Rode o programa resultante na imagem fantom.pgm e verifique se as saídas são diferentes daquelas obtidas usando núcleo de Scharr.

Exercício: O gradiente pode ajudar a detectar círculos na imagem, como na figura ao lado. Explique que propriedade o gradiente possui que pode facilitar a detecção de círculos. A técnica que detecta círculos em imagens é a transformada de Hough.



Exercício: O programa abaixo aplica filtros de Sobel 3×3 na imagem *circulo.png*, obtendo os componentes do gradiente *ox.png* e *oy.png* (figura 21).

```
//exercicio sobel.cpp - 2024
#include "procimagem.h"

int main() {
    Mat_<float> a=imread("circulo.png",0);
    if (a.total()==0) erro("Erro leitura");
    Mat_<float> sx,sy,ox,oy;
    Sobel(a,sx,-1,1,0,3); ox=sx/4.0+128; imwrite("ox.png",ox);
    Sobel(a,sy,-1,0,1,3); oy=sy/4.0+128; imwrite("oy.png",oy);
    //Complete o programa aqui utilizando matrizes sx e sy
}
```

Complete o programa, fazendo operações de módulo pixel a pixel (abs), inverter sinal pap (- com um argumento), soma pap (+), subtração pap (- com dois argumentos), mínimo pap (min), máximo pap (max) e elevar/raiz (pow) com as matrizes sx e sy, para obter as imagens com:

- Todas as bordas verticais: ver_todo.png
- A borda vertical esquerda: ver_esq.png
- A borda vertical direita: ver_dir.png.
- Todas bordas horizontais: hor_todo.png
- A borda horizontal superior: hor_sup.png
- A borda horizontal inferior: hor_inf.png
- Módulo do gradiente: modulo.png

Nota: A função para elevar por 2 de C++ é pow(ent, 2, sai). Calcular a raiz quadrada é pow(ent, 0.5, sai).




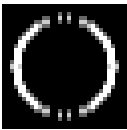
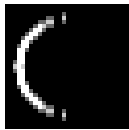
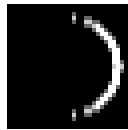
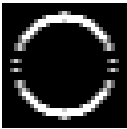

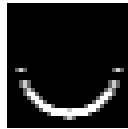
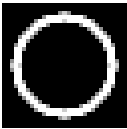
		
circulo.png	ox.png	oy.png
		
ver_todo.png	ver_esq.png	ver_dir.png
		
hor_todo.png	hor_sup.png	hor_inf.png
		
modulo.png		

Figura 21: Saídas esperadas do exercício.

7.2 Suavização da direção de gradiente

Considere o triângulo da figura 22a e o seu gradiente calculado na figura 22b. Gostaríamos que as direções dos vetores gradiente de um certo lado do triângulo fossem todos iguais. Mas isto não acontece, pois o lado do triângulo é formado por “escadinhas”, fazendo com que a direção do gradiente fique variando. O gradiente pode ser suavizado filtrando a imagem com gaussiana antes de calcular gradiente (figuras 22c e 22d).

O mesmo problema acontece ao calcular o gradiente de um círculo. A direção do gradiente não muda suavemente (figura 22e), por causa dos “degraus” nas bordas do círculo. Filtrando a imagem primeiro com o filtro gaussiana podemos obter gradiente onde a direção muda suavemente.

Como vimos, a função do OpenCV que aplica filtro gaussiana chama-se `GaussianBlur`. É possível obter o mesmo efeito que aplicar filtro gaussiana usando núcleos maiores (5×5 , 7×7 , etc.). Estes núcleos maiores podem ser obtidos a partir das expressões algébricas das derivadas da função gaussiana (figura 17).

Exercício: A seguinte sequência de comandos em Cekeikon aplica o filtro gaussiano com desvio-padrão 1 pixel na imagem `circulo.png`, calcula gradiente e armazena como imagem complexa em `circulo.img`. Depois, representa o campo vetorial do gradiente como flechas e armazena na imagem `circulof.png`.

```
kcek gradieng circulo.png circulo.txt 1
kcek campox circulo.txt circulof.png 100 15
```

Experimente rodar esta sequência de comandos. Mude os parâmetros, como o desvio-padrão do filtro gaussiano (deve ser >0). Visualize o gradiente de outras imagens, como `triang.png`.

Nota: Escreva “kcek gradieng” ou “kcek campox” para obter uma breve descrição dos parâmetros dos programas.

Nota: Os códigos desses programas estão em (...)/cekeikon5/cekeikon/kcek.

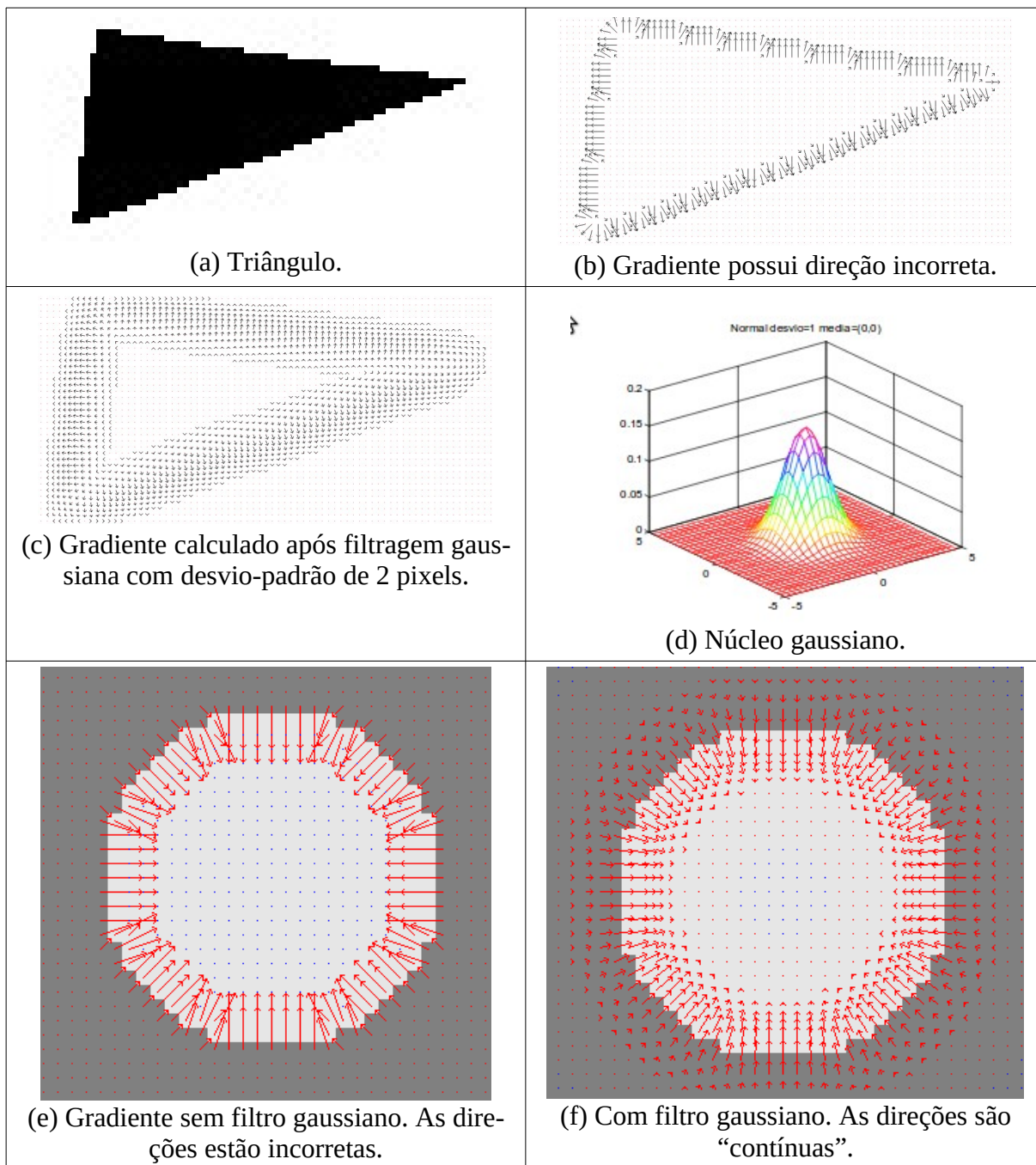
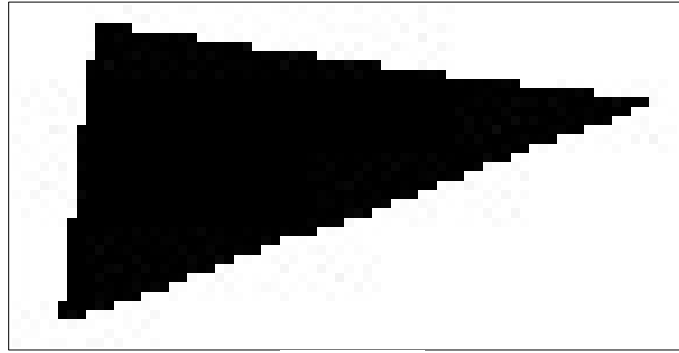
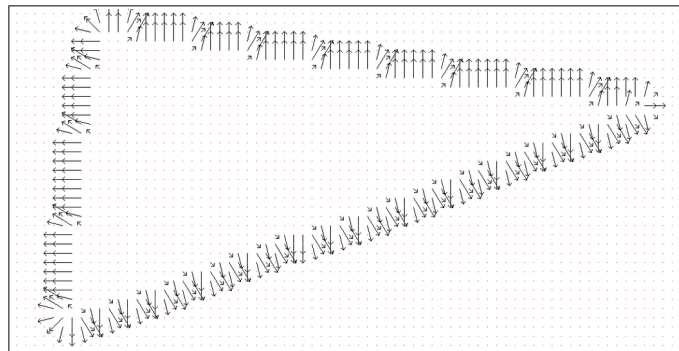


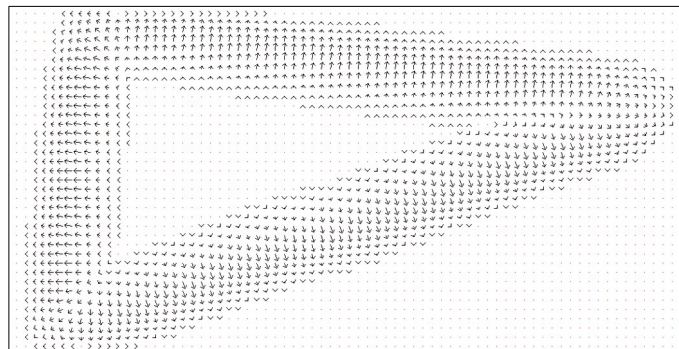
Figura 22: O gradiente pode ser melhorado suavizando a imagem de entrada com filtro gaussiano.



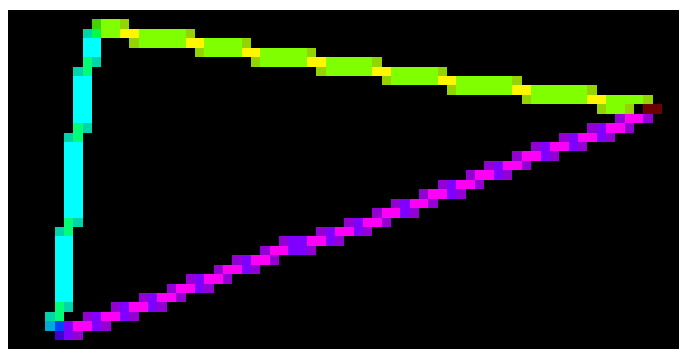
(a) triang.png



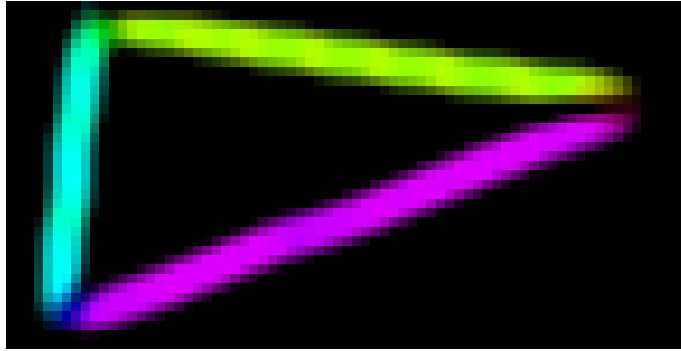
(b) triang0.png. Sem filtragem gaussiana. Alguns vetores têm direção errada.



(c) triang2.png. Filtragem gaussiana sigma=desvio=2 pixels. Menos erro na direção.



(d) triang0.img (sem filtro gaussiano) usando brilho como magnitude e cor como direção do gradiente. A direção do gradiente não é constante numa aresta.



(e) triang2.img (com filtro gaussiano de $\sigma=2$ pixels) usando brilho como magnitude e cor como direção do gradiente. A direção do gradiente é mais ou menos constante numa aresta.

Figura 23:

Exercício: Foram geradas 10 imagens circ00.png a circ09.png executando o programa abaixo. Essas imagens estão em [filtconv.zip](#).

```
//circulo.cpp
#include <cekeikon.h>
int main() {
    int nl=200, nc=200, ming=40, maxg=70, minp=2, maxp=7;
    for (int i=0; i<10; i++) {
        Mat_<GRAY> a(nl,nc,255);
        circulo(a, maxg+rand()%(nl-2*maxg), maxg+rand()%(nc-2*maxg), ming+rand()%(maxg-ming), 80, -1);
        for (int j=0; j<20; j++) {
            circulo(a, maxp+rand()%(nl-2*maxp), maxp+rand()%(nc-2*maxp), minp+rand()%(maxp-minp), 80, -1);
        }
        mostra(a);
        imp(a,format("circ%02d.png",i));
    }
}
```

Cada uma dessas imagens contém um círculo grande com raio entre 40 e 70 pixels e 20 círculos pequenos com raios entre 2 e 7 pixels. Duas dessas imagens estão abaixo:

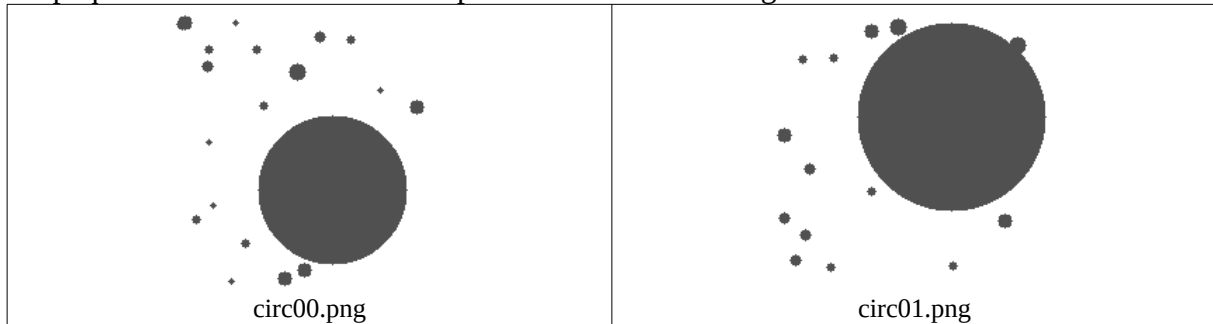


Figura 24: Exemplos de imagens circ??.png

Faça um programa que, dada uma imagem como acima, localiza o centro do círculo grande usando o fato de que o centro do círculo se encontra na intersecção das retas-suportes dos gradientes. Você deve usar obrigatoriamente esta ideia. Marque o centro do círculo com algum sinal bem visível, por exemplo, um “X” vermelho.

Nota: A biblioteca OpenCV possui uma função chamada HoughCircles que detecta círculos. Não pode usar esta função ou transformada de Hough para detectar círculos pronta de qualquer outra biblioteca.

8. Convolução e correlação

Convolução e correlação estão intimamente ligadas ao filtro linear. Em Visão Computacional, os três termos costumam ser usados como sinônimos.

Convolução:

A convolução discreta de duas imagens $f(x,y)$ e $h(x,y)$ é denotada por $f(x,y)*h(x,y)$ e definida pela expressão:

$$f(x,y)*h(x,y) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m,n)h(x-m,y-n)$$

Aqui, estamos considerando que as duas imagens estão preenchidas com zeros fora dos respectivos domínios. Convolução equivale a filtro linear onde o núcleo foi rotacionado por 180 graus (repare que na definição do filtro linear na pg. 14 usamos o sinal de “+” em vez de “-”).

Correlação:

A correlação discreta de duas imagens $f(x,y)$ e $h(x,y)$ de tamanhos $M \times N$ é denotada por $f(x,y) \circ h(x,y)$ e definida pela expressão:

$$f(x,y) \circ h(x,y) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f^*(m,n)h(x+m,y+n)$$

Novamente, estamos considerando que as duas imagens estão preenchidas com zeros fora dos respectivos domínios. f^* é o conjugado complexo de f . Evidentemente, se f for real, $f^*=f$. Assim, a correlação é igual ao filtro linear para imagens reais.

Nota: Se f e h são iguais, a operação chama-se auto-correlação. Se são diferentes, chama-se correlação cruzada.

9. Teorema da convolução

O teorema da convolução diz:

$$f(x,y)*h(x,y) \Leftrightarrow F(u,v) \cdot H(u,v) \quad \text{e} \quad f(x,y) \cdot h(x,y) \Leftrightarrow F(u,v)*H(u,v)$$

onde:

- $F = \text{DFT}(f)$ (transformada discreta de Fourier)
- f e h são matrizes reais.
- F e H são matrizes complexas, transformações de Fourier de f e h .
- $*$ indica convolução.
- $F(u,v) \cdot H(u,v)$ indica a multiplicação pixel a pixel (produto de Hadamard).

Não irei explicar este teorema detalhadamente. O que interessa para nós é que a convolução pode ser calculada rapidamente usando transformada rápida de Fourier (FFT) e a sua inversa (IFFT). Esta propriedade é útil para aplicar filtro linear com núcleo grande. O manual do OpenCV diz que a função `filter2D` usa FFT/IFFT para acelerar cálculo de filtros com núcleo 11×11 ou maior.

[PSI5790-2025 aula 2 parte 1 – Fim]

Referências:

[WikiIntegral] https://en.wikipedia.org/wiki/Summed-area_table.

[Perreault2007] Simon Perreault and Patrick Hebert, “Median Filtering in Constant Time,” IEEE T. IMAGE PROCESSING, vol. 16; no. 9, pp 2389-2394, 2007.

[Gonzalez2002] Gonzalez, Rafael C., and Richard E. Woods. “Digital Image Processing” (2002).