

# Worksheet3\_pdfversion

Creation date: 2024-11-18 20:45

Modification date: Monday 18th November 2024 20:45:19

PDF Ref:

## Section 1: Filtragem no domínio espacial

**1. Construa uma função para remover o ruído de uma imagem utilizando o filtro de média. Deve usar a média de vizinhança para obter a imagem filtrada (sem usar o conceito de convolução com kernel). Experimente com vizinhanças de 3x3, 5x5 e 9x9 e comente os resultados. Aplique às imagens "head\_gaussian\_noise.png" e "head\_saltpepper\_noise.png".**

Código:

```
exercise1_ws3_avgFilter

1 void exercise1_ws3(const string& imagePath, const string& imageName, int kernel) {
2
3     Mat image = imread(imagePath, IMREAD_GRAYSCALE);
4     Mat paddedImage;
5     int pad = kernel/2; // pad is equal to the kernel/2
6     //creates a padded image
7     // 4x pad because its top, bottom, left and right
8     // border_constant => puts all 0 (default).
9     copyMakeBorder(image, paddedImage, pad, pad, pad, pad, BORDER_CONSTANT);
10
11    // 1 channel image to serve as final reference
12    Mat filterImage(image.rows, image.cols, CV_8UC1, Scalar(0));
13
14    // loops through each pixel of the padded image, excluding the padding borders.
15    for (int r = pad; r < paddedImage.rows - pad; r++) {
16        for (int c = pad; c < paddedImage.cols - pad; c++) {
17            int sum = 0;
18            // loops through the kernel region centered at (r, c).
19            // accumulates the sum of all pixel intensities in the kernel.
20            for (int kx = -pad; kx <= pad; kx++) { // go through x
21                for (int ky = -pad; ky <= pad; ky++) { // go through y
22                    sum += paddedImage.at<uchar>(r + kx, c + ky);
23                }
24            }
25            // avg calculation of pixels for the output image
26            filterImage.at<uchar>(r - pad, c - pad) = static_cast<int>(round(float(sum)) / (kernel *
27 kernel));
28        }
29    }
30
31    Mat combinedImage;
32    hconcat(image, filterImage, combinedImage); // image side by side
33
34    imshow(imageName, combinedImage);
35    waitKey(0);
}
```

## Referências:

Tipicamente aplicado a ruídos gaussianos.

Quanto maior a vizinhança, maior a suavização.

Ruído gaussiano → menor variação nos pixels da imagem, corresponde a uma distribuição gaussiana.

Ruído Salt and Pepper → variações acentuadas, valores muito diferentes dos pixels da vizinhança em termos de intensidade. Outliers.

O que é esperado de uma filtragem média a um ruído gaussiano:

- Boa filtragem



Pasted image 20241118184944.png#center

O que é esperado de uma filtragem média a um ruído salt and pepper:

- Fraca filtragem



Pasted image 20241118185608.png#center

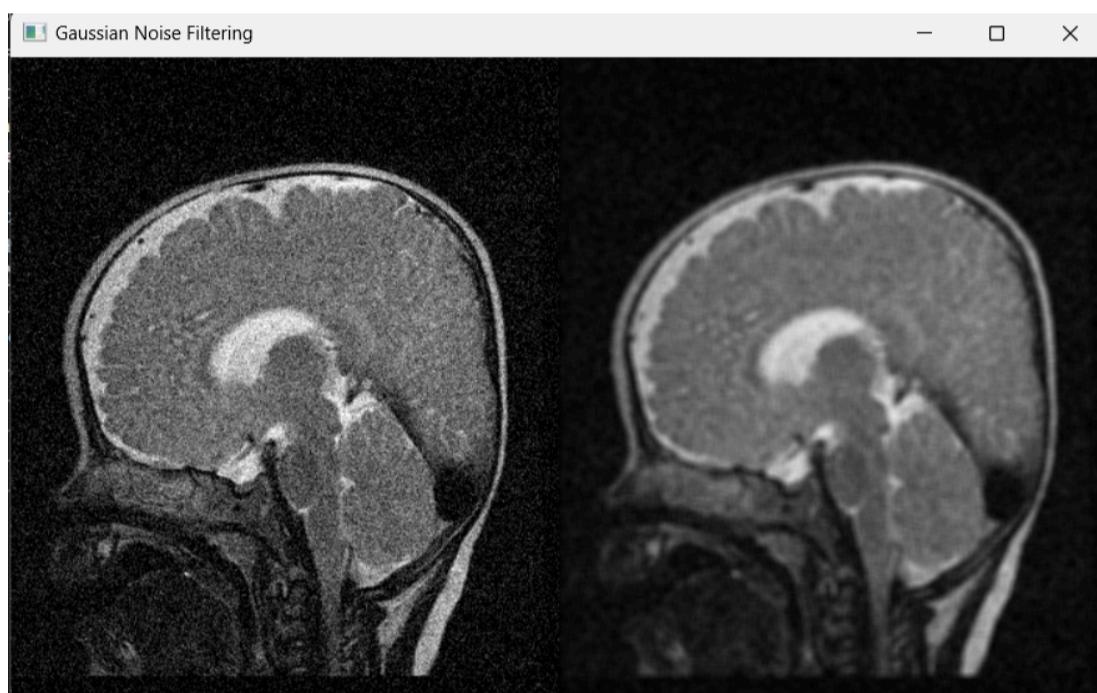
*Resultados:*

- Head Gaussian Noise Kernel 3×3:



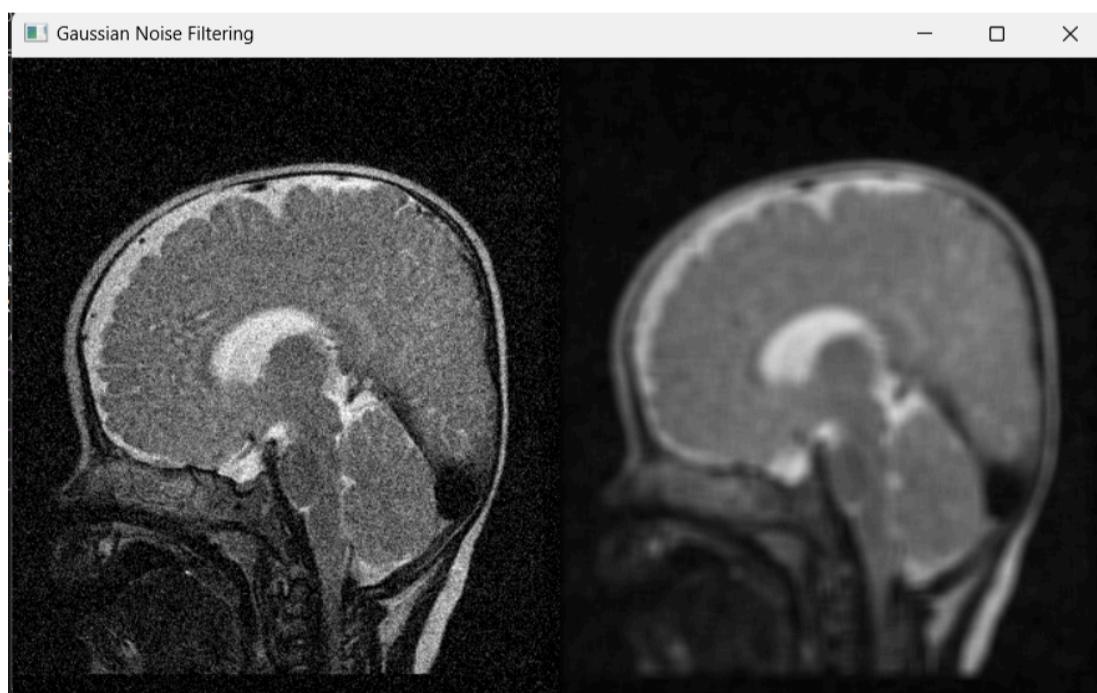
Pasted image 20241128161029.png#center

- Head Gaussian Noise Kernel 5×5:



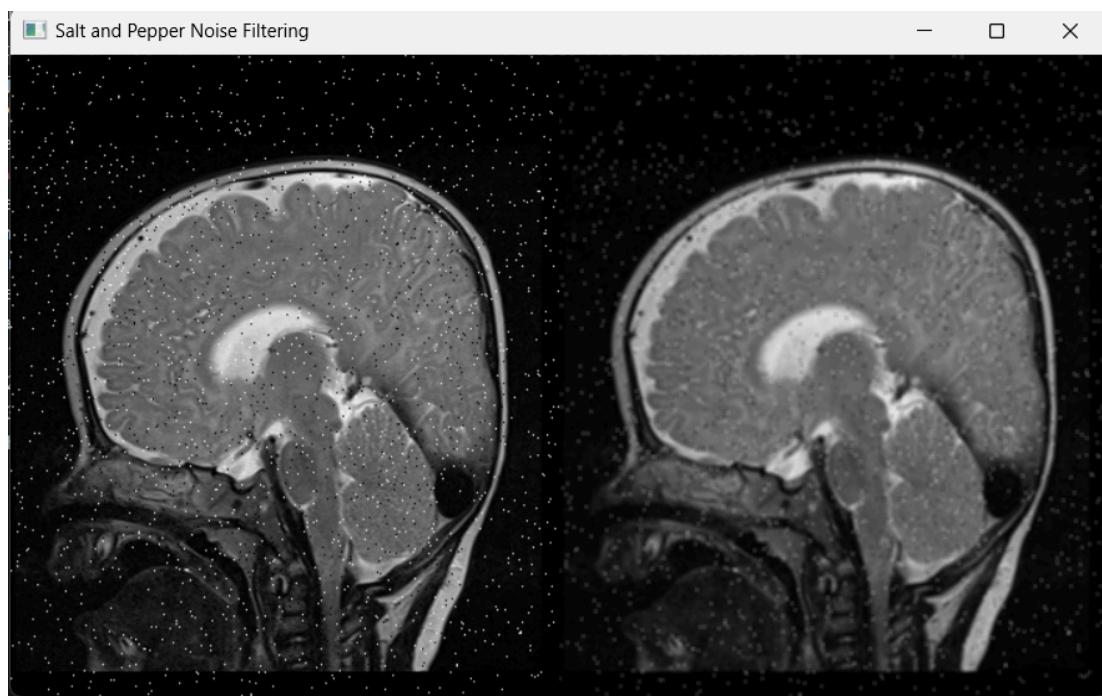
Pasted image 20241128161302.png#center

- Head Gaussian Noise Kernel 9×9:



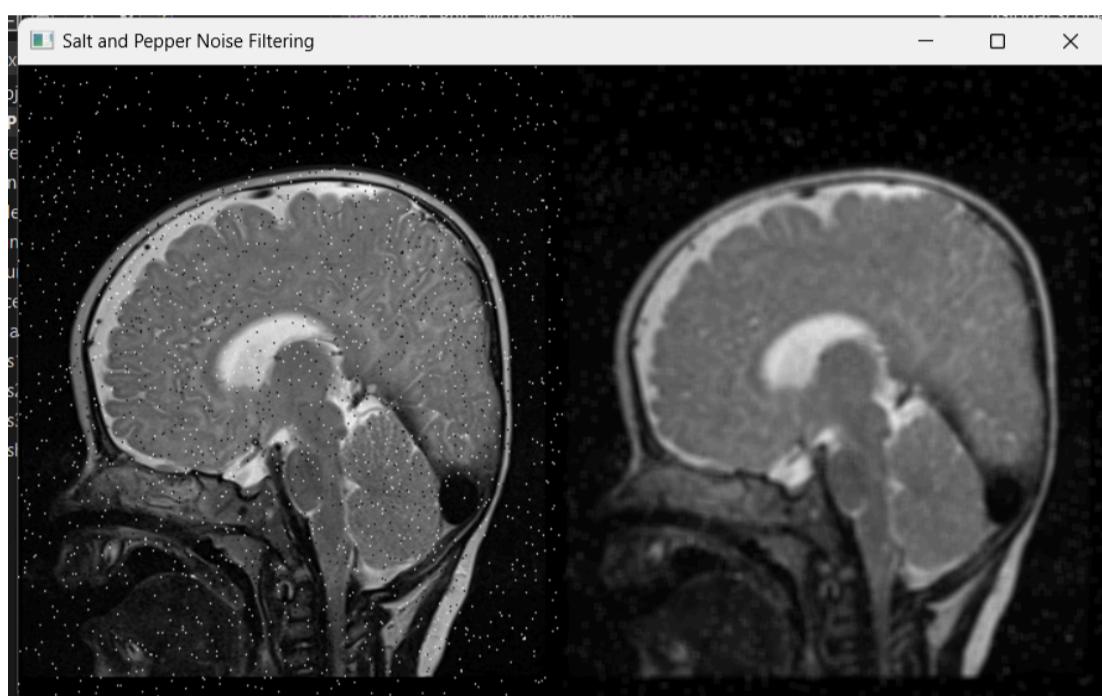
Pasted image 20241128161147.png#center

- Head Salt and Pepper Noise 3×3:



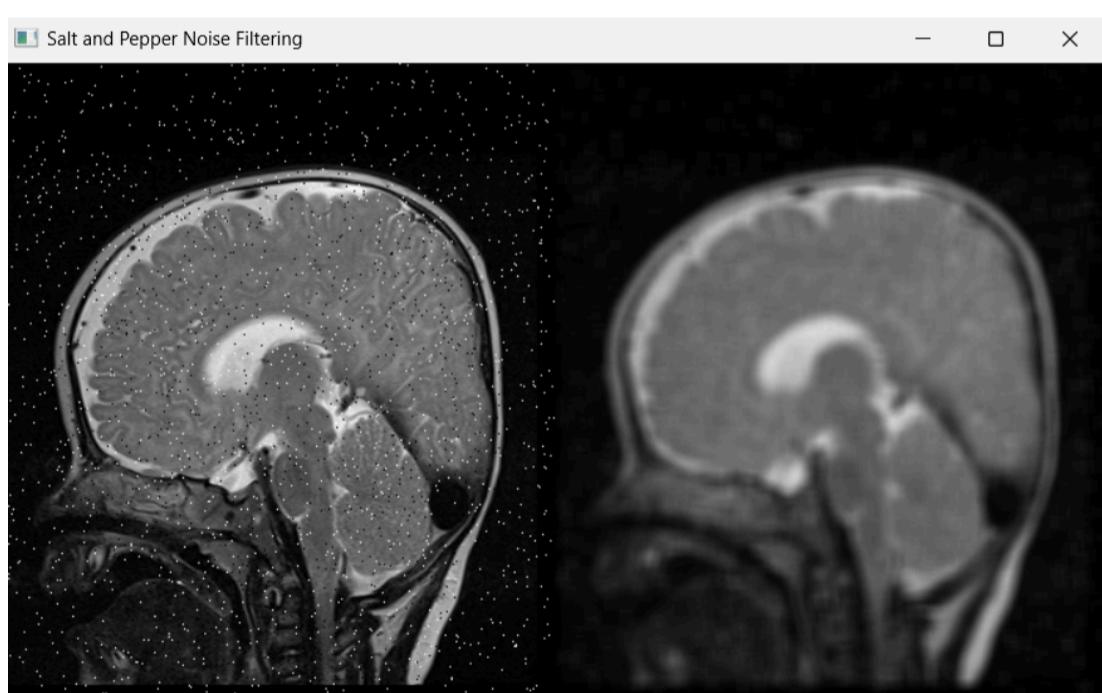
Pasted image 20241128162058.png#center

- Head Salt and Pepper Noise 5×5:



Pasted image 20241128162151.png#center

- Head Salt and Pepper Noise 9×9:



Pasted image 20241128162213.png#center

### Código:

- Inicialmente é realizada a leitura da imagem (em grayscale) conforme o path enviado pelo sistema de userInput (imagem com ruído gaussiano → 0 e ruído salt and pepper → 1).
- A essa imagem é transformada numa imagem com padding (bordas a 0), para permitir realizar as operações nos cantos da imagem, através da função copyMakeBorder(). A imagem resultante é guardada na imagem tipo Mat paddedImage.
- Uma imagem do mesmo tamanho e tipo CV\_8UC1 (grayscale) é criada para servir como output final após a operação.
- O programa vai percorrer toda a imagem com padding, e realizar a soma de todos os valores num loop interno que percorre todo o kernel.
- Para cada iteração/cada pixel do loop que percorre a imagem com padding, a respetiva média é calculada e arredondada através da fórmula: **Sum / Kernel x Kernel**
- E o resultado do cálculo que corresponde à nova intensidade de cada pixel, vai ser copiado para a nova imagem filtrada em grayscale.
- No final apenas existe uma concatenação das duas imagens (original e filtrada), para poder visualizar as duas lado a lado num imshow().

### Gaussiano:

- Inicialmente o filtro é aplicado com 3 tipos de kernel diferentes, a uma imagem com ruido gaussiano (ruído que segue uma distribuição gaussiana na intensidade dos pixels).
- A filtragem suaviza a imagem através da média dos pixels da vizinhança, conforme o kernel aplicado, para todos os pixels da imagem. Logicamente, quanto maior o kernel, mais suavizada será a imagem e consequentemente o ruído.
- A suavização do ruído acontece porque as variações de intensidade não são muito intensas em relação aos pixels próximos/na vizinhança (ao contrário do ruído salt and pepper).
- A suavização também pode chegar a um ponto em que a imagem começa a ficar demasiado suavizada e os detalhes e arestas desaparecem, o que não é o pretendido. O pretendido é suavizar o suficiente para obter uma imagem que ainda seja possível identificar as áreas de interesse.
- O resultado entre os 3 testes é o esperado conforme as referências, onde o ruído é atenuado à medida que o kernel aumenta.

### Salt and Pepper:

- Aplicados filtros com 3 kernels diferentes a uma imagem com ruído salt and pepper, que teoricamente será uma má abordagem em termos de filtragem.
- Facilmente é possível verificar pelos outputs que todas as filtragens não foram suficientes para eliminar o ruído, que é composto de grandes variações de intensidade (outliers pretos e brancos de 0 e 255, respetivamente).
- O que acontece realmente é que o ruído dos pixels com ruído vai "espalhar" para a vizinhança, resultando em zonas meias "nebulosas" ou manchadas.
- A única maneira de eliminar este ruído com um filtro destes, seria utilizar um kernel muito grande, que é o mesmo que estragar completamente a imagem, deixando de existir zonas de interesse visíveis. A única solução eficaz é utilizar um filtro de mediana, para eliminar estes outliers.

**2. Construa uma função para remover o ruído de uma imagem utilizando o filtro de mediana. Experimente com vizinhanças de 3x3, 5x5 e 9x9 e comente os resultados. Aplique às imagens "head\_gaussian\_noise.png" e "head\_saltpepper\_noise.png". Comente as diferenças em relação à questão 1.**

Código:

```
exercise2_ws3_medianFilter

1 void exercise2_ws3(const string& imagePath, const string& imageName, int kernel) {
2
3     Mat image = imread(imagePath, IMREAD_GRAYSCALE);
4     Mat paddedImage;
5     int pad = kernel / 2; // pad is equal to the kernel/2
6     //creates a padded image
7     // 4x pad because its top, bottom, left and right
8     // border_constant => puts all 0 (default).
9     copyMakeBorder(image, paddedImage, pad, pad, pad, pad, BORDER_CONSTANT);
10    imshow("Padded image", paddedImage);
11
12    // 1 channel image to serve as final reference
13    Mat filterImage(image.rows, image.cols, CV_8UC1, Scalar(0));
14
15    // loops through each pixel of the padded image, excluding the padding borders.
16    for (int r = pad; r < paddedImage.rows - pad; r++) {
17        for (int c = pad; c < paddedImage.cols - pad; c++) {
18            vector<int> kernelValues; // To store the kernel values
19            // loops through the kernel region centered at (r, c).
20            // accumulates the sum of all pixel intensities in the kernel.
21            for (int kx = -pad; kx <= pad; kx++) { // go through x
22                for (int ky = -pad; ky <= pad; ky++) { // go through y
23                    kernelValues.push_back(paddedImage.at<uchar>(r + kx, c + ky));
24                }
25            }
26            // Sort the kernel values to find the median
27            sort(kernelValues.begin(), kernelValues.end());
28
29            // Calculate the median
30            int median = 0;
31            int kernelSizeSquared = kernel * kernel;
32            if (kernelSizeSquared % 2 == 0) { // check if it's an even number
33                // takes in consideration the two values in the middle / because its not odd
34                median = (kernelValues[kernelSizeSquared / 2 - 1] + kernelValues[kernelSizeSquared / 2]) /
35                2;
36            }
37            else { // check if it's an odd number
38                median = kernelValues[kernelSizeSquared / 2]; // its the middle element
39            }
40            // Assign the median value to the output image pixels / without padding (r - pad, c - pad)
41            filterImage.at<uchar>(r - pad, c - pad) = static_cast<uchar>(median);
42        }
43    Mat combinedImage;
44    hconcat(image, filterImage, combinedImage); // image side by side
45
46    imshow(imageName, combinedImage);
47    waitKey(0);
48 }
```

## Referências:

Tipicamente aplicado a ruídos gaussianos.

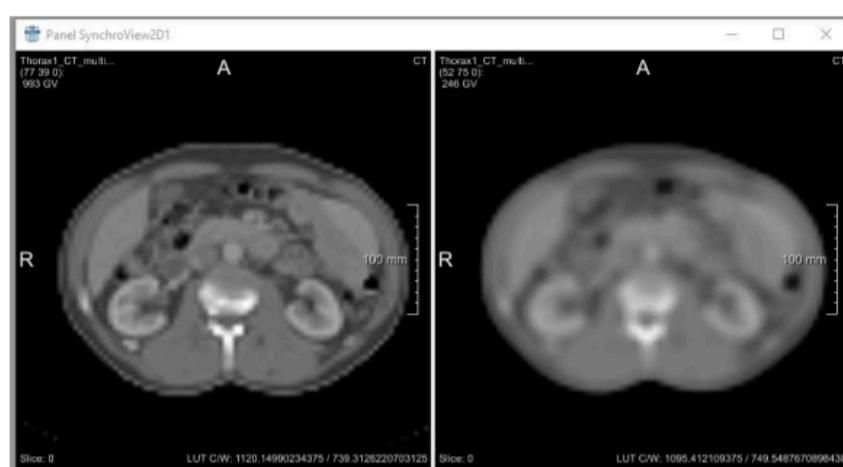
Quanto maior a vizinhança, maior a suavização.

Ruído gaussiano → menor variação nos pixels da imagem, corresponde a uma distribuição gaussiana.

Ruído Salt and Pepper → variações acentuadas, valores muito diferentes dos pixels da vizinhança em termos de intensidade. Outliers.

O que é esperado de uma filtragem média a um ruído gaussiano:

- Boa filtragem



Pasted image 20241118184944.png#center

O que é esperado de uma filtragem média a um ruído salt and pepper:

- Fraca filtragem



Pasted image 20241118185608.png#center

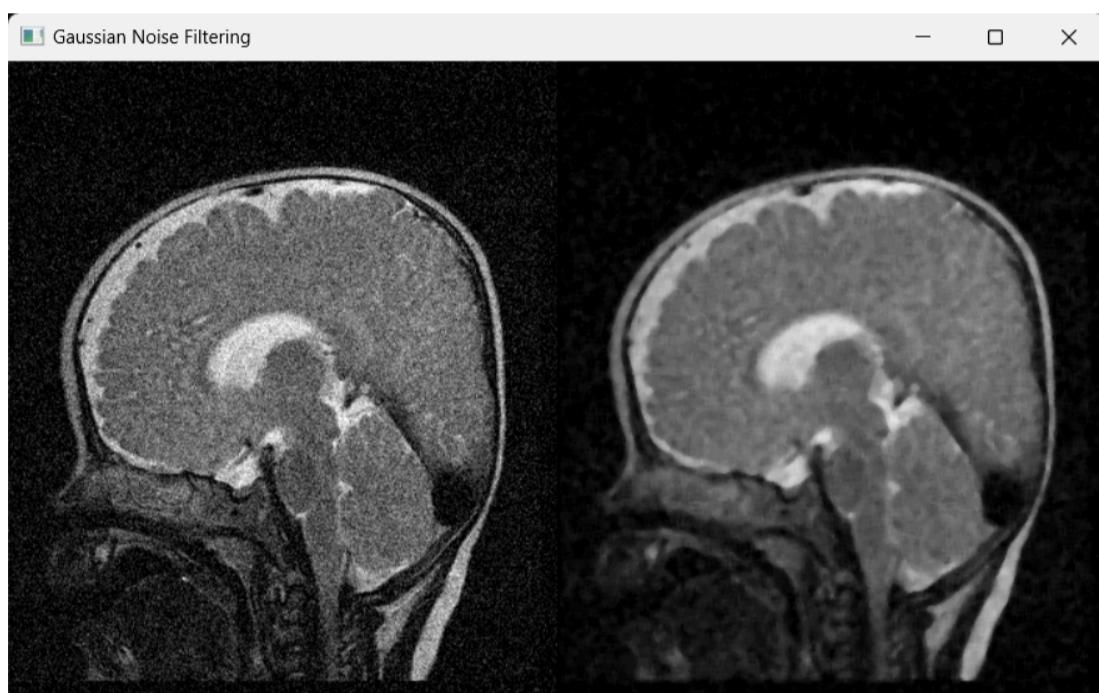
*Resultados:*

- Head Gaussian Noise Kernel 3×3:



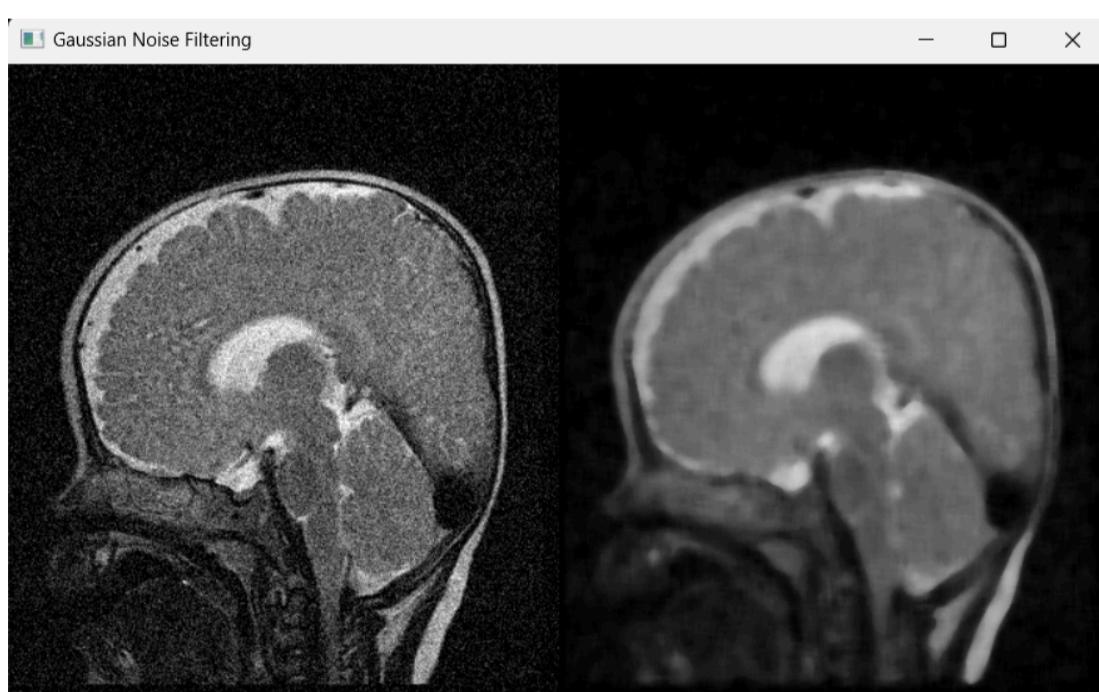
Pasted image 20241129100638.png#center

- Head Gaussian Noise Kernel 5×5:



Pasted image 20241129100717.png#center

- Head Gaussian Noise Kernel 9×9:



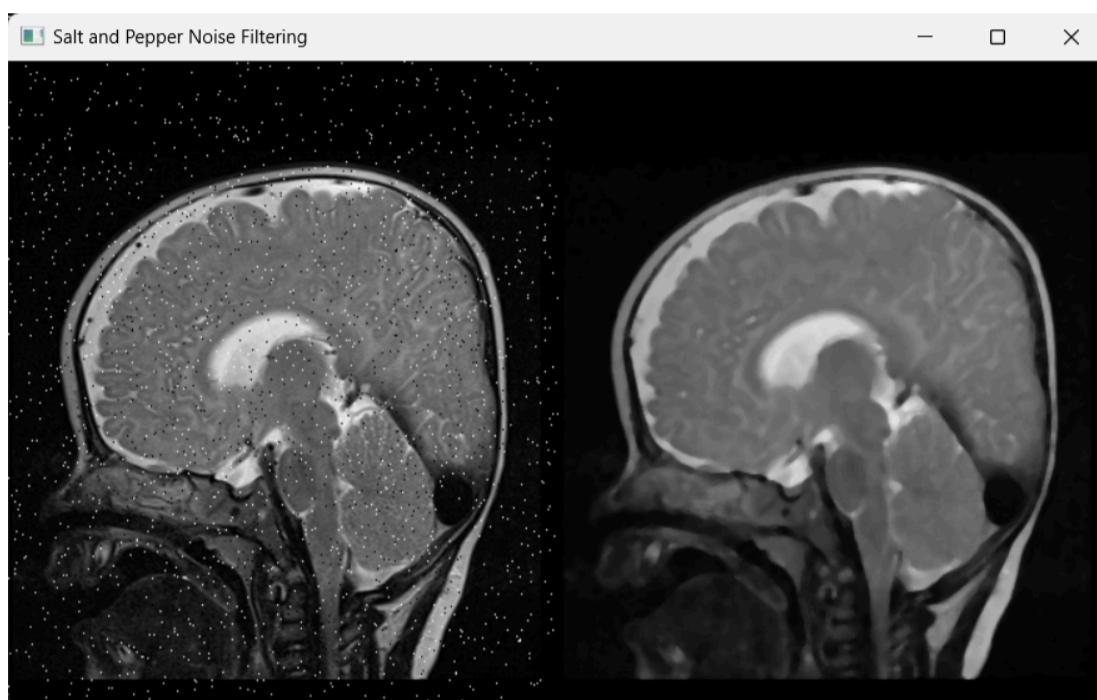
Pasted image 20241129100745.png#center

- Head Salt and Pepper Noise Kernel 3×3:



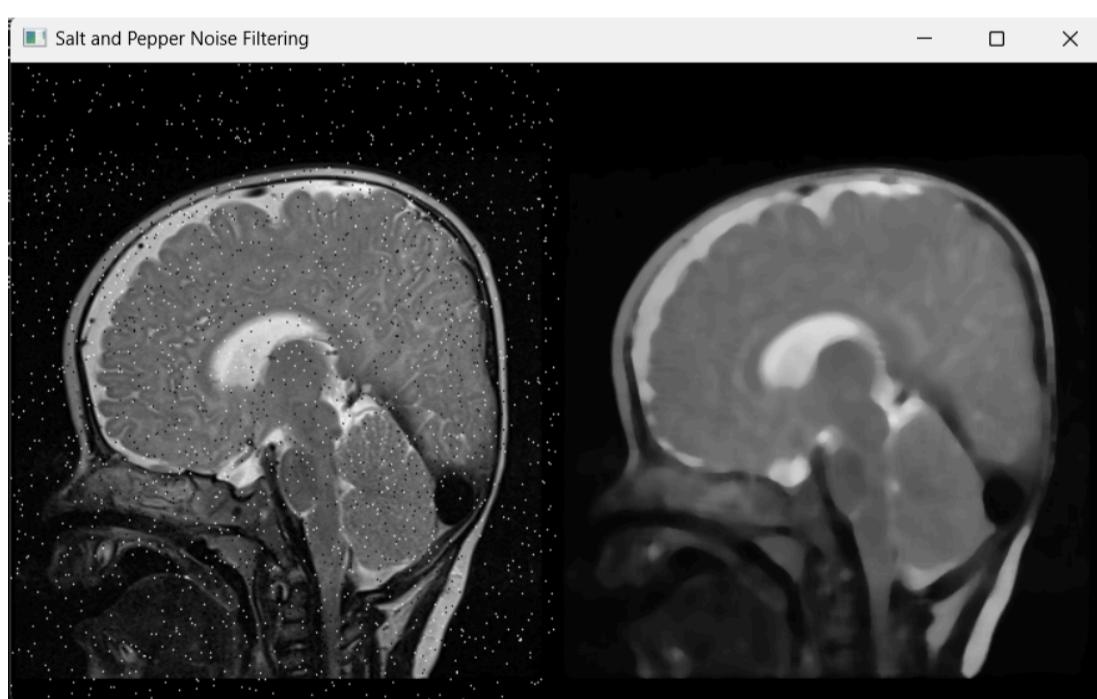
Pasted image 20241129100816.png#center

- Head Salt and Pepper Noise Kernel 5×5:



Pasted image 20241129100838.png#center

- Head Salt and Pepper Noise Kernel 9×9:



Pasted image 20241129100859.png#center

### **Código:**

- Inicia da mesma maneira que o exercício anterior em termos de criação de uma imagem com padding (da imagem original), e a imagem de output final, que vai receber a imagem transformada pela operação de filtragem.
- Os loops são os mesmos, que percorrem a imagem pixel a pixel, e fazem os cálculos no kernel.
- Só que desta vez é criado um vetor para cada pixel, que vai receber o valor de todas as intensidades do kernel.
- Os valores são posteriormente organizados por ordem, para de seguida calcular o resto e saber se o número é ímpar ou par (conforme se faz na mediana).
- Se for ímpar, o valor central é a nova intensidade do pixel que será transferido para a imagem de output, senão (se for par) os dois valores centrais são somados e divididos por 2.
- No final as imagens são concatenadas para ficar lado a lado e mostradas com imshow().

### **Gaussiano:**

- Tal como era esperado, o filtro não consegue remover um ruído gaussiano corretamente, visto que este filtro é especializado na eliminação de outliers, e os valores da intensidade nesse caso, não são tão dispares dos valores da vizinhança.
- Por isso, à medida que o kernel aumenta, a imagem fica cada vez mais suavizada mas o ruído continua aparente.
- Por isso, o filtro média é a melhor opção para um ruído gaussiano, como é possível verificar no exercício anterior.

### **Salt and Pepper:**

- Neste caso, o ruído é completamente removido, já que o filtro é o adequado para a remoção de valores de intensidade muito bruscos em relação à vizinhança, os tais outliers.
- Mesmo com um kernel baixo ( $3 \times 3$ ), a imagem fica sem qualquer ruído.
- O aumento do kernel apenas vai levar à suavização da imagem no geral.
- Por isso, o filtro mediana é a melhor opção para um ruído salt and pepper, como é possível verificar no exercício anterior.

### 3. Construa uma função que para construir um kernel gaussiano.

Código:

```
gaussianKernel

1 Mat exercise3_ws3(int kernelSize, int sigma){
2
3     int waitUser = 0;
4     int w = kernelSize / 2;
5
6     // CV_64F = double in cpp
7     Mat Gauss(kernelSize, kernelSize, CV_64F, Scalar(0));
8
9     // Fill the kernel using the Gaussian formula
10    for (int i = -w; i <= w; i++) {
11        for (int j = -w; j <= w; j++) {
12            double value = (1 / (2 * 3.14 * sigma * sigma)) *
13                exp(-((i * i + j * j) / (2 * sigma * sigma))); // gaussian formula
14            // the value is applied to the matrix created
15            Gauss.at<double>(i + w, j + w) = value;
16        }
17    }
18
19    // Normalize the kernel
20    Gauss /= sum(Gauss)[0]; // Normalize so the sum of all elements equals 1
21
22    return Gauss;
23 }
```

## Referências:

<https://www.geeksforgeeks.org/gaussian-filter-generation-c/>

In a **Gaussian kernel**, the **sigma ( $\sigma$ )** represents the **standard deviation** of the Gaussian distribution. It plays a crucial role in determining the shape and spread of the kernel. Here's a detailed explanation:

**What is Sigma ( $\sigma$ )?**

- Sigma is a parameter of the Gaussian function that controls the width of the bell curve.
- It is directly proportional to the spread of the kernel: the larger the sigma, the wider and flatter the curve; the smaller the sigma, the narrower and sharper the curve.

**Gaussian Function (1D):**

The formula for a 1D Gaussian function is:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

In 2D (for images):

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Here,  $x$  and  $y$  are spatial coordinates, and  $\sigma$  determines how quickly the Gaussian decays as you move away from the center.

Pasted image 20241202104610.png#center

## Effects of Changing Sigma:

1. **Small Sigma ( $\sigma \rightarrow 0$ ):**
  - The kernel becomes very narrow, focusing only on the immediate neighborhood.
  - Results in **sharp details** but less smoothing or blurring in the image.
  - Useful for applications where small-scale details matter.
2. **Large Sigma ( $\sigma \rightarrow \infty$ ):**
  - The kernel becomes broader and flatter.
  - Results in more **smoothing or blurring** as the influence of neighboring pixels extends over a larger area.
  - Useful for reducing noise and large-scale smoothing.

## Applications in Image Processing:

- **Blurring:** A larger sigma results in more significant blurring.
- **Edge Detection:** The choice of sigma affects the scale of edges detected when using techniques like the **Laplacian of Gaussian (LoG)** or **Difference of Gaussians (DoG)**.
- **Feature Extraction:** Sigma helps in multi-scale analysis, as features of different sizes can be detected using different values of sigma.

Pasted image 20241202104625.png#center

Normalization in the context of a Gaussian kernel (or any kernel used in image processing) ensures that the sum of all elements in the kernel equals 1. This is an important step because it prevents the kernel from inadvertently altering the overall brightness or intensity of the image during convolution.

Pasted image 20241202105738.png#center

*Resultados:*

- Kernel de  $3 \times 3$  com sigma de 6 (normalizado)

```
Choose kernel size:  
3  
Choose sigma value:  
6  
Gaussian Kernel:[0.1111111111111111, 0.1111111111111111, 0.1111111111111111;  
0.1111111111111111, 0.1111111111111111, 0.1111111111111111;  
0.1111111111111111, 0.1111111111111111, 0.1111111111111111]  
Press any number to go back.  
|
```

The  $3 \times 3$  Gaussian kernel with  $\sigma = 6$ , normalized, is:

$$\begin{bmatrix} 0.1101 & 0.1116 & 0.1101 \\ 0.1116 & 0.1132 & 0.1116 \\ 0.1101 & 0.1116 & 0.1101 \end{bmatrix}$$

**Código:**

- Inicialmente é criada uma matriz com tamanho de 64 bits, equivalente a um double em C.
- Dois for loops percorrem essa matriz e preenchem os valores com o resultado da fórmula gaussiana.
- O tamanho da matriz segue o tamanho do kernel que o utilizador escolheu.
- O kernel é normalizado no final (usado para prevenir alterações no brilho geral da imagem).

**4. Construa uma função para filtrar uma imagem utilizando o princípio da convolução com kernels. Aplique a função à imagem "head\_gaussian\_noise.png" usando um kernel gaussiano.**

Código:

```
convolveGaussian

1 void exercise4_ws3(const string& imagePath, const string& imageName, Mat kernel, int kernelSize) {
2
3     Mat image = imread(imagePath, IMREAD_GRAYSCALE);
4
5     Mat paddedImage;
6     int pad = kernelSize / 2;
7     copyMakeBorder(image, paddedImage, pad, pad, pad, pad, BORDER_CONSTANT, Scalar(0));
8
9     Mat filterImage(image.rows, image.cols, CV_32F, Scalar(0)); // Ensure float type
10
11    for (int r = pad; r < paddedImage.rows - pad; r++) {
12        for (int c = pad; c < paddedImage.cols - pad; c++) {
13            float sum = 0.0;
14
15            for (int kx = -pad; kx <= pad; kx++) {
16                for (int ky = -pad; ky <= pad; ky++) {
17                    float kernelValue = kernel.at<float>(kx + pad, ky + pad);
18                    float pixelValue = (float)paddedImage.at<uchar>(r + kx, c + ky);
19                    sum += static_cast<float>(pixelValue) * kernelValue;
20                }
21            }
22            filterImage.at<float>(r - pad, c - pad) = sum;
23        }
24    }
25
26    Mat displayImage;
27    filterImage.convertTo(displayImage, CV_8U); // Convert for display
28
29    Mat combinedImage;
30    hconcat(image, displayImage, combinedImage);
31
32    imshow(imageName, combinedImage);
33    waitKey(0);
34 }
```

## Referências:

### 1. Width of the Bell Curve

- Sigma ( $\sigma$ ) controls the spread or width of the Gaussian curve.
- A smaller  $\sigma$ :
  - Results in a narrower peak with most values concentrated near the center.
  - Strongly emphasizes the central pixel and its immediate neighbors.
  - Leads to less blurring (sharper image details remain).
- A larger  $\sigma$ :
  - Produces a broader, flatter curve with values spread over a larger area.
  - Includes contributions from pixels farther from the center.
  - Results in more blurring (details are smoothed out).

Pasted image 20241129125319.png#center

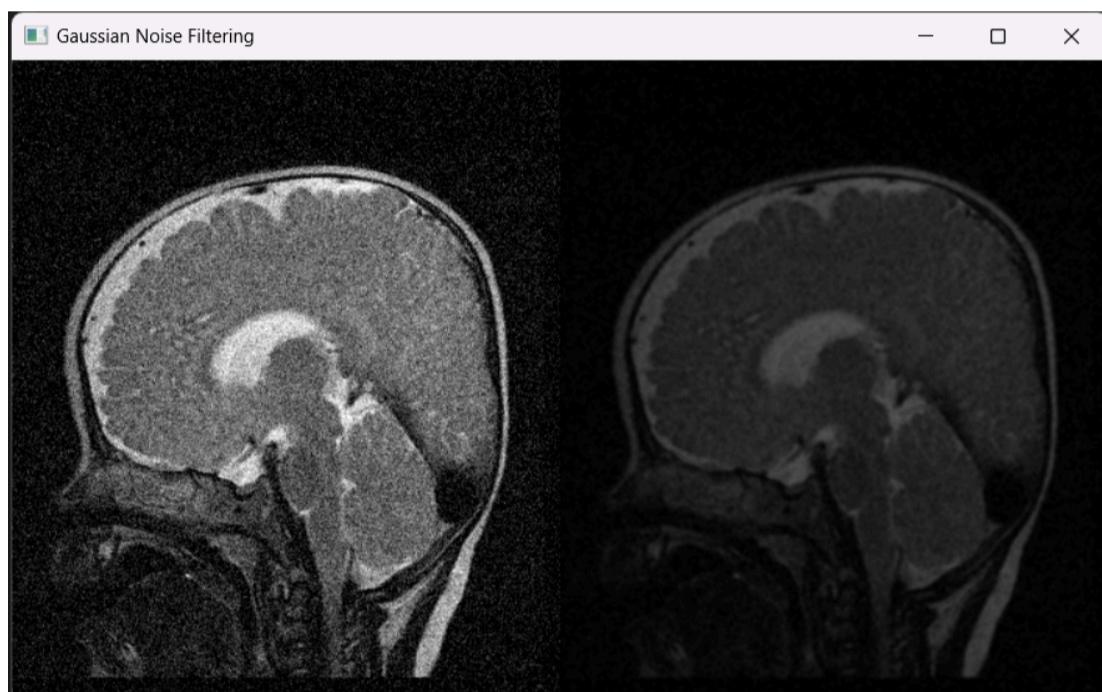
**Resultados:**

- Kernel 3x3 sigma 1:



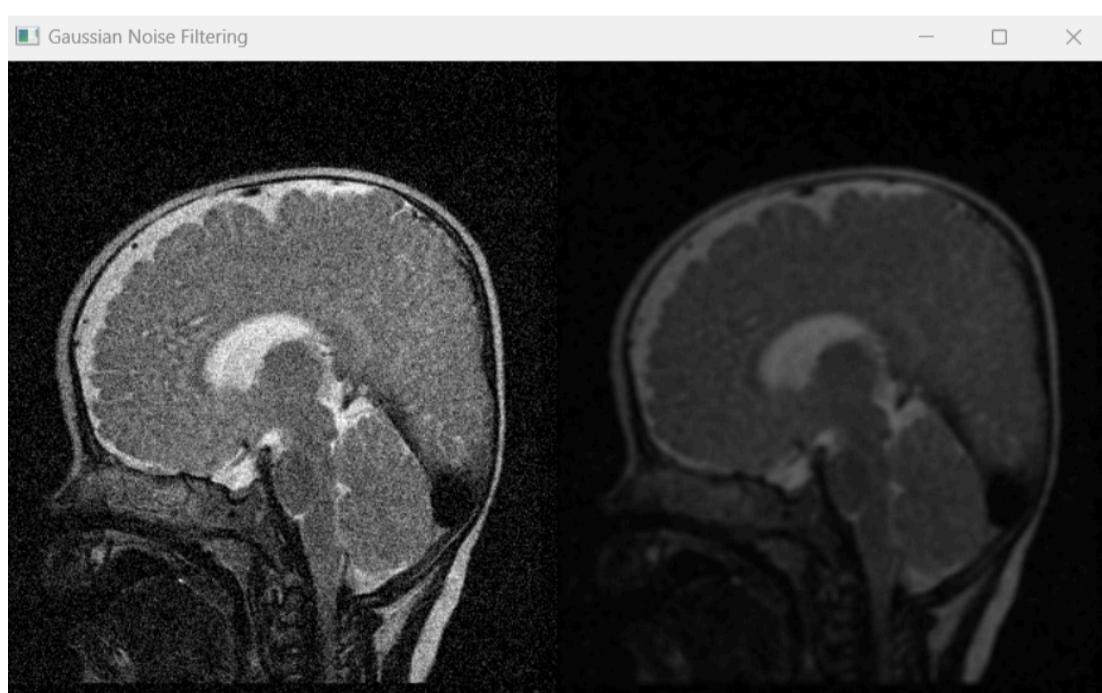
Pasted image 20241204151448.png#center

- Kernel 3x3 sigma 2:



Pasted image 20241204151403.png#center

- Kernel 5x5 sigma 3:



Pasted image 20241204151741.png#center

### **Código:**

- Inicialmente a função do exercício anterior é chamada para criar o kernel gaussiano normalizado.
- De seguida, o programa percorre toda a imagem com padding e realiza uma convolução através do kernel gaussiano, e guarda toda a informação de intensidade numa nova imagem, tipo float.

### **Ruído:**

- No primeiro exemplo (kernel  $3 \times 3$  com sigma 1), é possível notar uma ligeira suavização do ruído entre a imagem de output e a original.
- Apesar da suavização, o filtro Gaussiano com este kernel preserva bem as estruturas principais da imagem.
- Um sigma de 1 aumenta a suavização, expandindo a influência de cada pixel no processo de filtragem. Isto vai ajudar a eliminar os ruídos de intensidade moderada, mas pode começar a desfocar algumas bordas e detalhes menores da imagem.
- Por isso, o resultado é uma imagem ligeiramente esbatida e com claridade, porque o kernel é pequeno e o sigma também é.
- No exemplo seguinte o valor do sigma foi aumentado para 2, e é possível verificar que a imagem fica mais escura, o que já era de esperar devido à maior dispersão dos valores na curva gaussiana.
- No último exemplo, o output está ligeiramente mais esbatido devido ao aumento do kernel. Um kernel maior abrange mais pixels ao redor de cada ponto, resultando numa suavização mais extensa e redução do ruído numa área maior (perda de detalhes).

**5. Detete os contornos da imagem "head\_clean.png" utilizando o filtro de Sobel. Deve apresentar os resultados das derivadas em cada direção, bem como o resultado final.**

*Código:*

### sobelFilteringWithLowPass

```
1 void exercise5_ws3(const string& imagePath, const string& imageName) {
2
3     Mat image = imread(imagePath, IMREAD_GRAYSCALE);
4     Mat paddedImage;
5     int pad = 3 / 2; // Kernel size / 2
6     copyMakeBorder(image, paddedImage, pad, pad, pad, pad, BORDER_CONSTANT);
7
8     // Intermediate matrices for Sobel filter results
9     Mat filterImageHorizontal(image.rows, image.cols, CV_32F, Scalar(0));
10    Mat filterImageVertical(image.rows, image.cols, CV_32F, Scalar(0));
11    Mat filterImageMagnitude(image.rows, image.cols, CV_32F, Scalar(0));
12
13    // Sobel kernels
14    float kernelx[3][3] = { {1, 0, -1},
15                           {2, 0, -2},
16                           {1, 0, -1} };
17
18    float kernely[3][3] = { {1, 2, 1},
19                           {0, 0, 0},
20                           {-1, -2, -1} };
21
22    for (int r = pad; r < paddedImage.rows - pad; r++) {
23        for (int c = pad; c < paddedImage.cols - pad; c++) {
24            float sum_x = 0, sum_y = 0; // Sums for both axes
25
26            for (int kx = -pad; kx <= pad; kx++) { // Kernel x
27                for (int ky = -pad; ky <= pad; ky++) { // Kernel y
28                    float pixel_value = static_cast<float>(paddedImage.at<uchar>(r + kx, c + ky));
29                    sum_x += pixel_value * kernelx[kx + pad][ky + pad]; // Convolution with Sobel X
30                    sum_y += pixel_value * kernely[kx + pad][ky + pad]; // Convolution with Sobel Y
31                }
32            }
33
34            // Sobel results
35            filterImageHorizontal.at<float>(r - pad, c - pad) = sum_x;
36            filterImageVertical.at<float>(r - pad, c - pad) = sum_y;
37
38            // Magnitude (combined Sobel X and Y results)
39            float magnitude = sqrt(sum_x * sum_x + sum_y * sum_y);
40            filterImageMagnitude.at<float>(r - pad, c - pad) = magnitude;
41        }
42    }
43
44    // Low-pass filter results
45    Mat lowPassSobelHorizontal = lowpassFilter(image, filterImageHorizontal, pad);
46    Mat lowPassSobelVertical = lowpassFilter(image, filterImageVertical, pad);
47    Mat lowPassSobelMagnitude = lowpassFilter(image, filterImageMagnitude, pad);
48
49    // Normalize low-pass results for visualization
50    Mat lowPassHorizontalDisplay, lowPassVerticalDisplay, lowPassMagnitudeDisplay;
51    normalize(lowPassSobelHorizontal, lowPassHorizontalDisplay, 0, 255, NORM_MINMAX, CV_8UC1);
52    normalize(lowPassSobelVertical, lowPassVerticalDisplay, 0, 255, NORM_MINMAX, CV_8UC1);
53    normalize(lowPassSobelMagnitude, lowPassMagnitudeDisplay, 0, 255, NORM_MINMAX, CV_8UC1);
54
55    // Combine images for visualization
56    Mat combinedImage, combinedImage2, combinedImage3;
57    hconcat(image, lowPassHorizontalDisplay, combinedImage);
58    hconcat(combinedImage, lowPassVerticalDisplay, combinedImage2);
59    hconcat(combinedImage2, lowPassMagnitudeDisplay, combinedImage3);
60
61    // Add labels to identify each section
```

```

62     putText(combinedImage3, "Original", Point(50, 50), FONT_HERSHEY_SIMPLEX, 1, Scalar(255, 255, 255), 2);
63     putText(combinedImage3, "Horizontal", Point(450, 50), FONT_HERSHEY_SIMPLEX, 1, Scalar(255, 255, 255),
64     2);
65     putText(combinedImage3, "Vertical", Point(800, 50), FONT_HERSHEY_SIMPLEX, 1, Scalar(255, 255, 255),
66     2);
67
68     // Display the final combined image
69     imshow(imageName, combinedImage3);
70     waitKey(0);
71 }

```

### *lowpassfilter*

```

1 Mat lowpassFilter(Mat& originalImage, Mat& filteredImage, int pad) {
2     Mat paddedImage;
3     Mat lowPass(originalImage.rows, originalImage.cols, CV_32F, Scalar(0));
4     copyMakeBorder(filteredImage, paddedImage, pad, pad, pad, pad, BORDER_CONSTANT, Scalar(0));
5
6     for (int r = pad; r < paddedImage.rows - pad; r++) {
7         for (int c = pad; c < paddedImage.cols - pad; c++) {
8             float sum = 0.0f;
9
10            for (int kx = -pad; kx <= pad; kx++) {
11                for (int ky = -pad; ky <= pad; ky++) {
12                    sum += paddedImage.at<float>(r + kx, c + ky);
13                }
14            }
15
16            // Average the kernel values
17            lowPass.at<float>(r - pad, c - pad) = sum / (3 * 3);
18        }
19    }
20
21    return lowPass;
22 }

```

## Referências:

x-direction kernel (the size is 3x3)

```
float kernelx[3][3] = {{-1, 0, 1},  
                        {-2, 0, 2},  
                        {-1, 0, 1}};
```

y-direction kernel

```
float kernely[3][3] = {{-1, -2, -1},  
                        {0, 0, 0},  
                        {1, 2, 1}};
```

To calculate the convolution at pixel (x,y), define a window of size equal to the kernel size (source code to calculate magnitude in x and magnitude in y are identical):

```
double magX = 0.0; // this is your magnitude  
  
for(int a = 0; a < 3; a++)  
{  
    for(int b = 0; b < 3; b++)  
    {  
        int xn = x + a - 1;  
        int yn = y + b - 1;  
  
        int index = xn + yn * width;  
        magX += image[index] * kernelx[a][b];  
    }  
}
```

Note that the input is a grayscale image and it can be represented as 1D array of double (This is just a trick, since a pixel value in coordinate (x,y) can be accessed with index = [x + y \* width] )

To calculate magnitude in pixel (x,y) given magX and magY :

**mag = sqrt( magX^2 + magY^2 )**

Share Edit Follow

edited Jul 7, 2018 at 14:29

Eric Lianoo

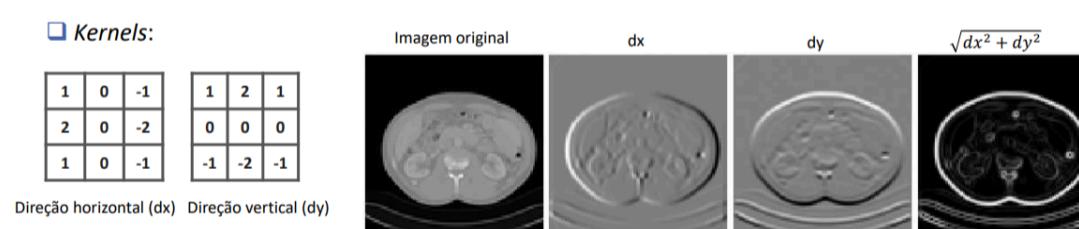
answered Jul 24, 2013 at 9:18

37080

Pasted image 20241202114117.png#center

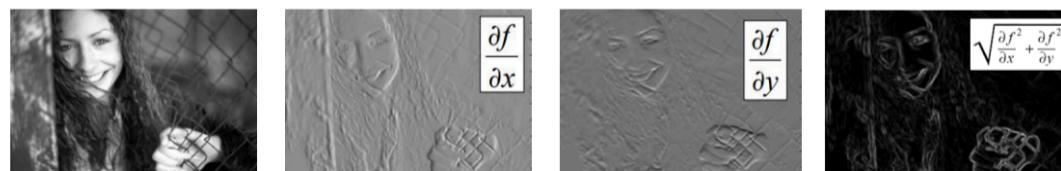
- ❑ Calcula o gradiente da imagem na direção X ou Y
  - Primeira derivada na respetiva direção
- ❑ Deteção de contornos
- ❑ Muitas vezes precedido por um filtro passa-baixo para eliminar ruído

Pasted image 20241202142414.png#center



Pasted image 20241202115858.png#center

### GRADIENTE



Pasted image 20241203152114.png#center

### 1. Necessidade de float:

- **Operações matemáticas precisas:**

Durante a aplicação dos kernels Sobel, os valores de cada pixel podem ser multiplicados por números positivos ou negativos, e os resultados acumulados (somados). O formato `uchar` (0-255) utilizado em imagens em escala de cinza não permite números negativos nem uma faixa de valores maior que 255.

Pasted image 20241203151747.png#center

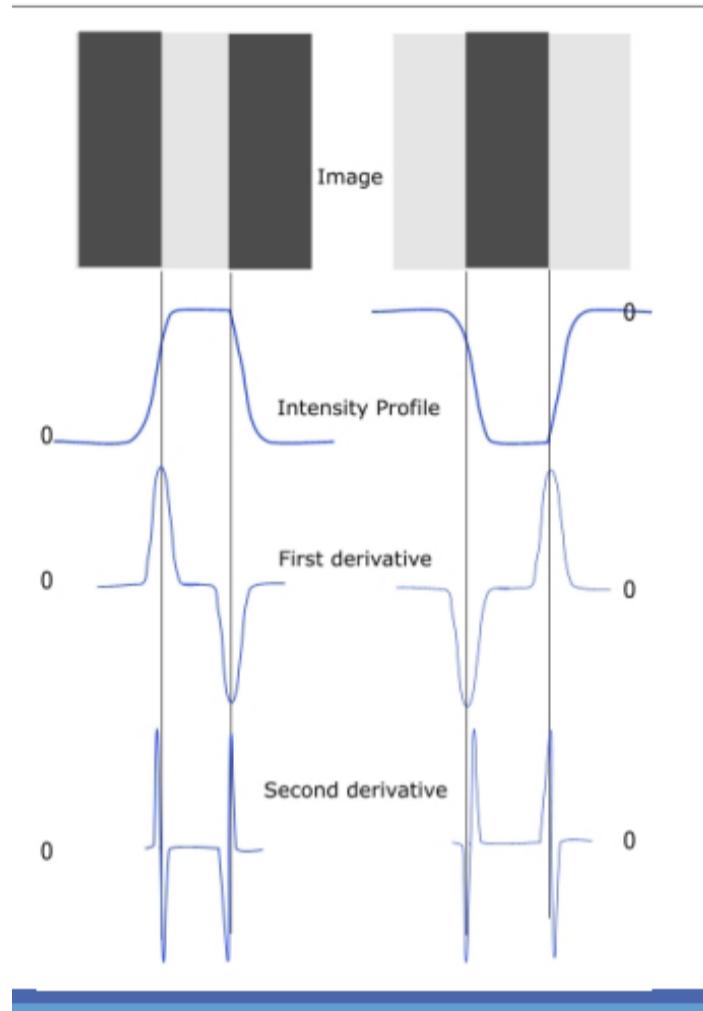
## 2. Necessidade de normalização:

- Transformar para um intervalo visualizável (0-255):

Depois das operações Sobel, os valores na imagem podem ser muito maiores ou menores do que o intervalo de 0-255, que é o intervalo aceito para exibição de imagens em OpenCV. A normalização ajusta os valores para esse intervalo, garantindo que os resultados possam ser exibidos corretamente.

$$\text{Valor Normalizado} = \frac{\text{Valor} - \text{Min}}{\text{Max} - \text{Min}} \times 255$$

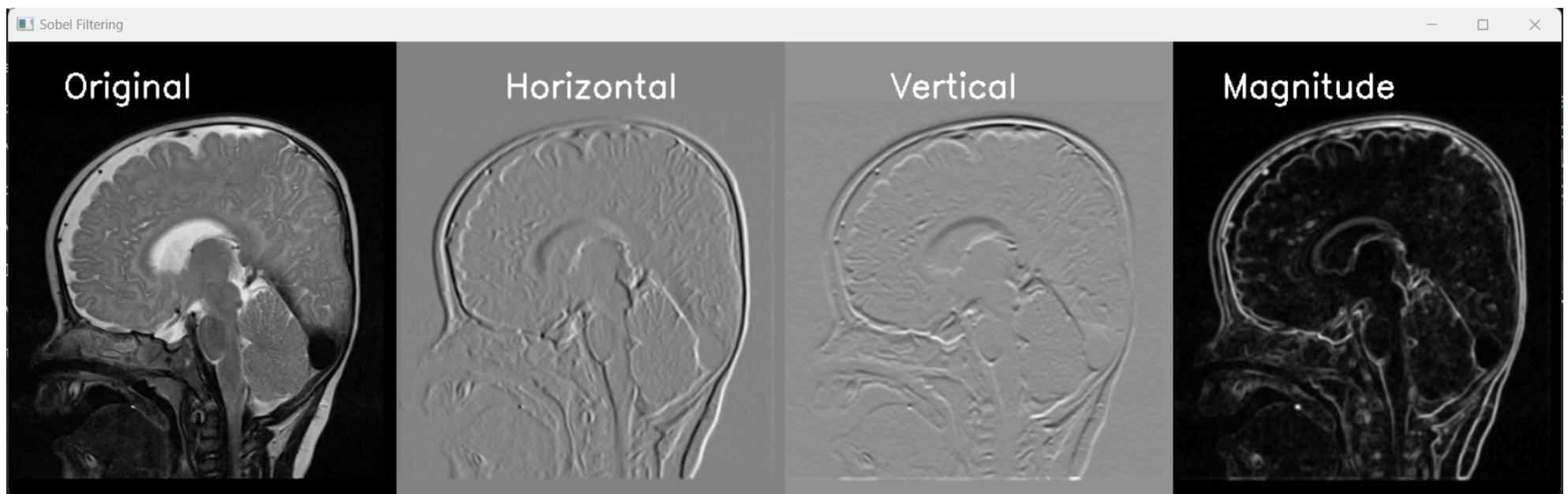
Pasted image 20241203151753.png#center



Pasted image 20241204145735.png#center

## Resultados:

- Com filtro passa baixo e normalização:



## Código:

- Realiza-se uma leitura de imagem original, e de seguida é criada uma cópia com padding.
- Criou-se também três matrizes tipo float (porque tratam com valores positivos e negativos) que vão receber os resultados das direções e da magnitude.
- Os kernels são os recomendados pelos slides, conforme é possível ver nas referências.
- De seguida é realizada uma convolução entre a imagem e os kernels, para x e y.
- E no final da operação a cada pixel, é realizada o cálculo da magnitude dos valores.
- Os respetivos valores de x, y e magnitude são copiados para as 3 matrizes criadas.
- Depois as imagens são filtradas para eliminar o ruído (por causa da recomendação nos slides), através de uma função criada para o propósito, que recebe como parâmetros a imagem original, a imagem a aplicar o filtro e o valor do pad (metade do kernel).
- Os valores são todos normalizados (para estarem num intervalo visualizável através do openCV).

## Sobel:

- Como é possível verificar na imagem de output, os contornos são detetados com sucesso, em ambas as direções, e na magnitude dos valores.
- Todas as zonas da imagem original onde existe uma transição rápida dos valores de intensidade, são "detetados" como contornos, em ambas as direções.
- **(No filtro de Sobel, a primeira derivada é calculada tanto na direção horizontal quanto vertical, identificando transições no brilho da imagem original.)**
- As zonas brancas nos contornos do crânio, são zonas onde existem essas mudanças abruptas de intensidade.
- Todas as imagens passaram por um filtro passa-baixo.

## 6. Faça a deteção dos contornos da imagem "head\_clean.png" usando o filtro de laplaciano.

Código:

laplaceFiltering

```
1 void exercise6_ws3(const string& imagePath, const string& imageName) {
2
3     Mat image = imread(imagePath, IMREAD_GRAYSCALE);
4     Mat paddedImage;
5     int pad = 3 / 2; // pad is equal to kernel/2
6     copyMakeBorder(image, paddedImage, pad, pad, pad, pad, BORDER_CONSTANT);
7     // 1-channel image to serve as final reference
8     Mat filterLaplace(image.rows, image.cols, CV_32F, Scalar(0));
9     // Laplacian kernel
10    float kernel[3][3] = {
11        {0, 1, 0},
12        {1, -4, 1},
13        {0, 1, 0}
14    };
15
16    for (int r = pad; r < paddedImage.rows - pad; r++) {
17        for (int c = pad; c < paddedImage.cols - pad; c++) {
18            float sum = 0.0f;
19
20            for (int kx = -pad; kx <= pad; kx++) { // go through x
21                for (int ky = -pad; ky <= pad; ky++) { // go through y
22                    float pixel_value = static_cast<float>(paddedImage.at<uchar>(r + kx, c + ky));
23                    sum += pixel_value * kernel[kx + pad][ky + pad]; // Convolution with Laplace
24                }
25            }
26
27            filterLaplace.at<float>(r - pad, c - pad) = sum;
28        }
29    }
30
31    // Low-pass filter
32    Mat lowPassLaplace = lowpassFilter(image, filterLaplace, pad);
33    // Normalize for display
34    Mat filterLaplaceDisplay, lowPassLaplaceDisplay;
35    normalize(filterLaplace, filterLaplaceDisplay, 0, 255, NORM_MINMAX, CV_8UC1);
36    normalize(lowPassLaplace, lowPassLaplaceDisplay, 0, 255, NORM_MINMAX, CV_8UC1);
37    // Combine images for visualization
38    Mat combinedImage, combinedImage2;
39    hconcat(image, filterLaplaceDisplay, combinedImage);
40    hconcat(combinedImage, lowPassLaplaceDisplay, combinedImage2);
41
42    // Add labels to identify each section
43    putText(combinedImage2, "Original", Point(50, 50), FONT_HERSHEY_SIMPLEX, 1, Scalar(255, 255, 255), 2);
44    putText(combinedImage2, "Laplace", Point(450, 50), FONT_HERSHEY_SIMPLEX, 1, Scalar(255, 255, 255), 2);
45    putText(combinedImage2, "Low pass", Point(800, 50), FONT_HERSHEY_SIMPLEX, 1, Scalar(0, 0, 0), 2);
46
47    // Display the final combined image
48    imshow(imageName, combinedImage2);
49    waitKey(0);
50 }
```

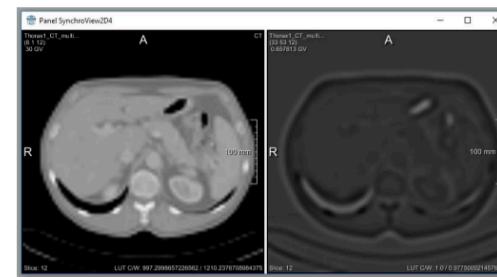
## lowpass

```
1 Mat lowpassFilter(Mat& originalImage, Mat& filteredImage, int pad) {
2
3     // Apply a low-pass filter to the horizontal Sobel output
4     Mat paddedImage;
5     Mat lowPass(originalImage.rows, originalImage.cols, CV_8UC1, Scalar(0));
6     copyMakeBorder(filteredImage, paddedImage, pad, pad, pad, pad, BORDER_CONSTANT, Scalar(0));
7
8     for (int r = pad; r < paddedImage.rows - pad; r++) {
9         for (int c = pad; c < paddedImage.cols - pad; c++) {
10            int sum = 0;
11
12            for (int kx = -pad; kx <= pad; kx++) {
13                for (int ky = -pad; ky <= pad; ky++) {
14                    sum += paddedImage.at<uchar>(r + kx, c + ky);
15                }
16            }
17
18            // Average the kernel values
19            lowPass.at<uchar>(r - pad, c - pad) = saturate_cast<uchar>(sum / (3 * 3));
20        }
21    }
22    return lowPass;
23 }
```

## Referências:

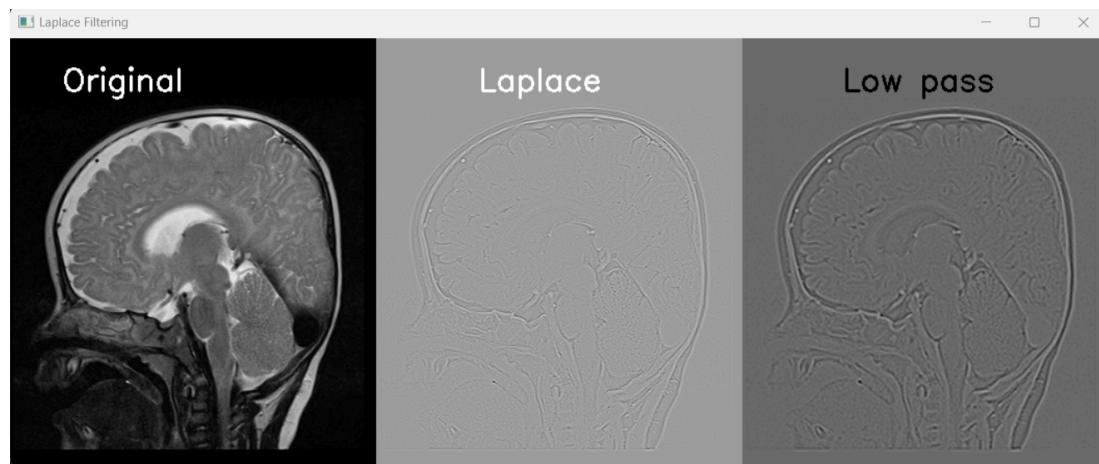
- Calcula a segunda derivada da imagem
- Realçar transições rápidas de intensidade
- Deteção de contornos
- Muitas vezes precedido por um filtro passa-baixo
- Kernel:*

0	1	0
1	-4	1
0	1	0



Pasted image 20241202142730.png#center

## Resultados:



Pasted image 20241204135709.png#center

## Código:

- Mesmo processo que o exercício anterior, com a mudança para o kernel Laplace.
- Agora não é possível detetar a direção, porque o cálculo é da segunda derivada.
- Convolução entre os valores da intensidade da imagem e do kernel, seguidos de um filtro passa baixo e normalização.

## Laplace:

- No caso do filtro Laplace, já não é possível retirar informações de direção, este filtro calcula agora a segunda derivada da imagem e serve para realçar as transições rápidas de intensidade.
- Tal como o Sobel, faz a deteção dos contornos, que são as áreas de transição.
- É possível ver a influencia que o filtro passa baixo teve na imagem, resultando num output mais fácil de visualizar, com menor claridade.