

Worksheet3_githubversion

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:

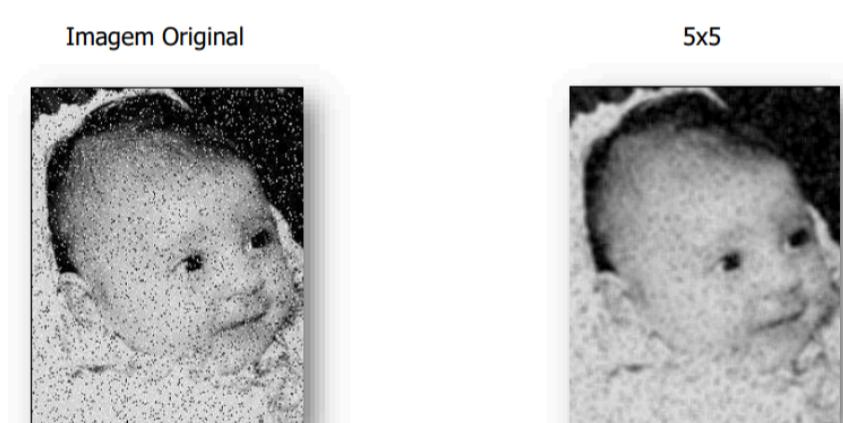
- Boa filtragem



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

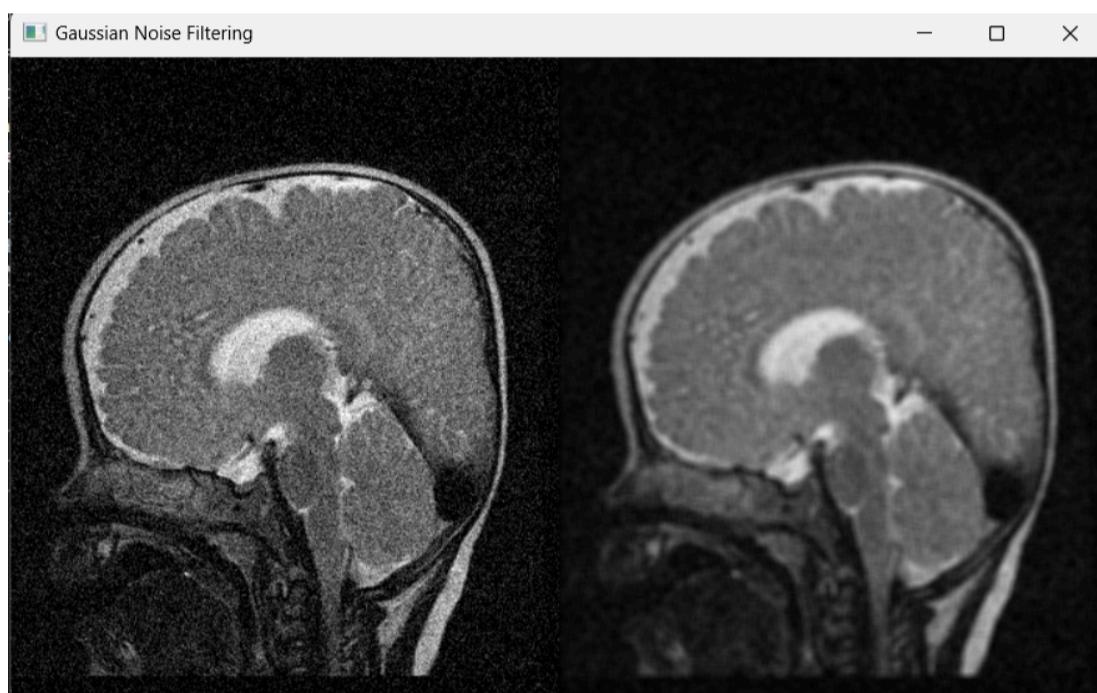
Resultados:

- Head Gaussian Noise Kernel 3×3:



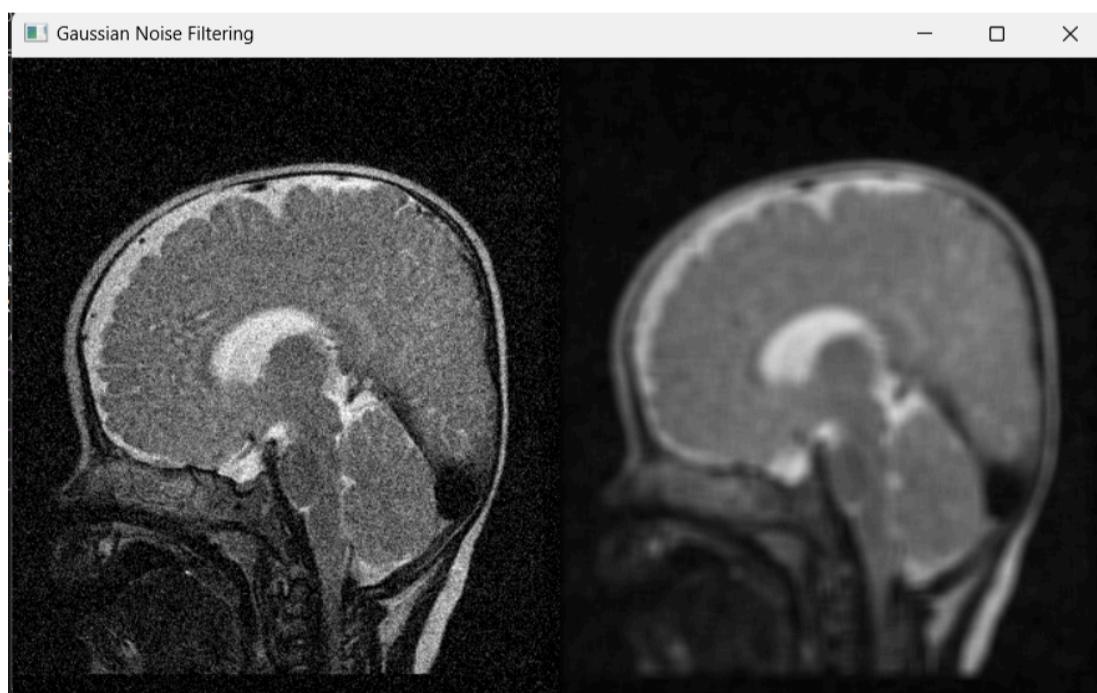
Pasted image 20241128161029.png#center

- 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:



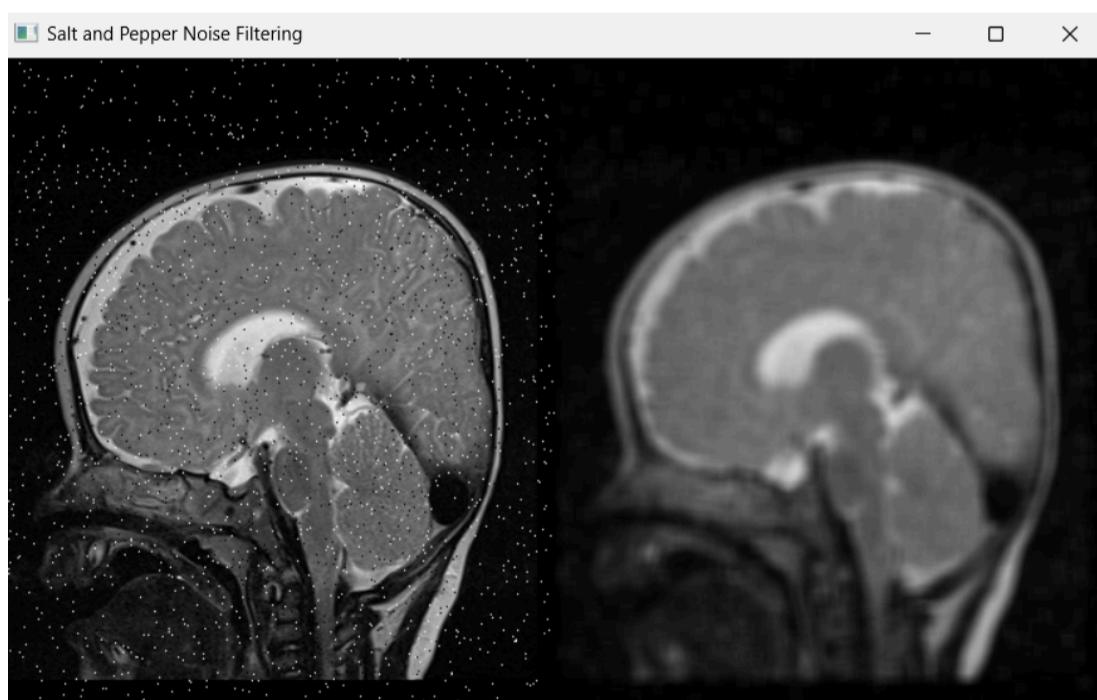
Pasted image 20241128162058.png#center

- 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.

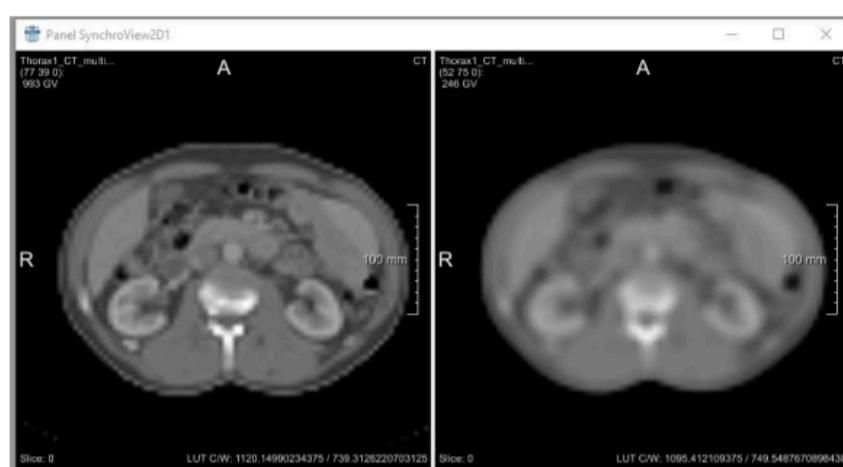
Quanto maior a vizinhança, maior a suavização.

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:



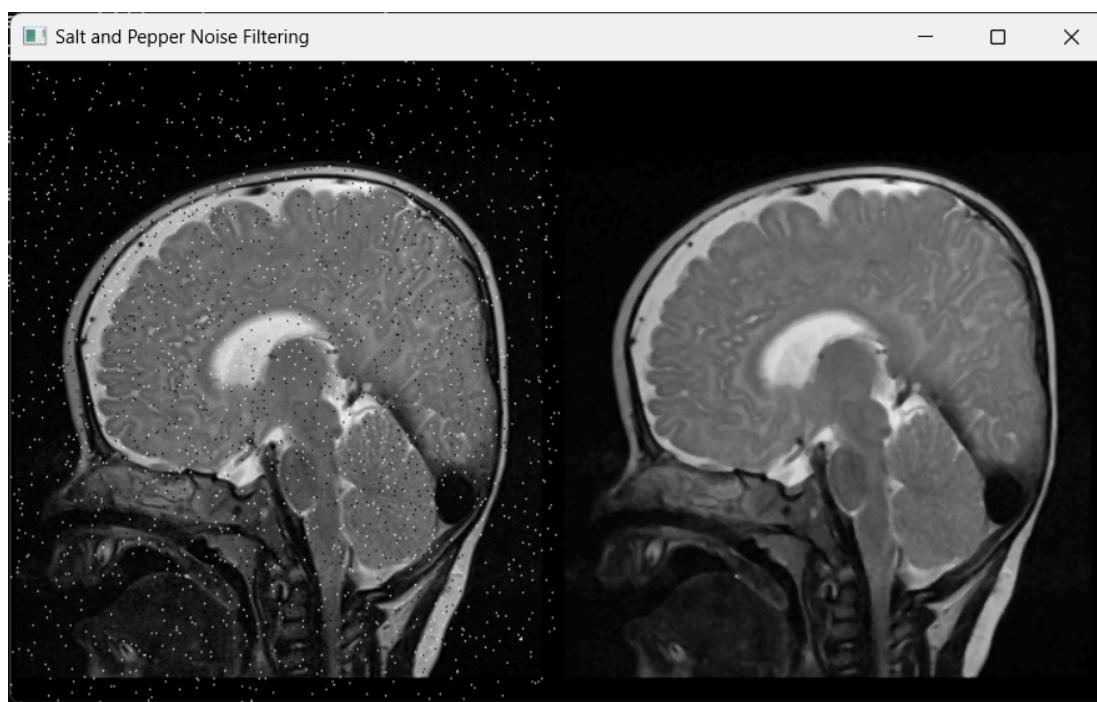
Pasted image 20241129100717.png#center

- Head Gaussian Noise Kernel 9×9:



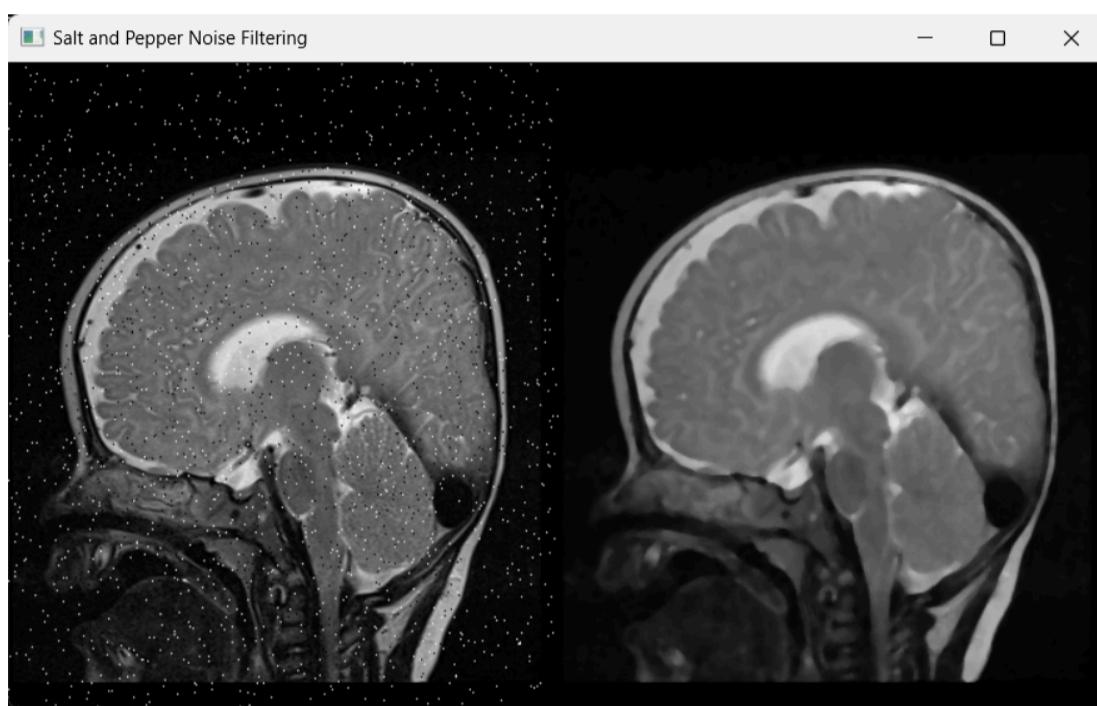
Pasted image 20241129100745.png#center

- Head Salt and Pepper Noise Kernel 3×3:



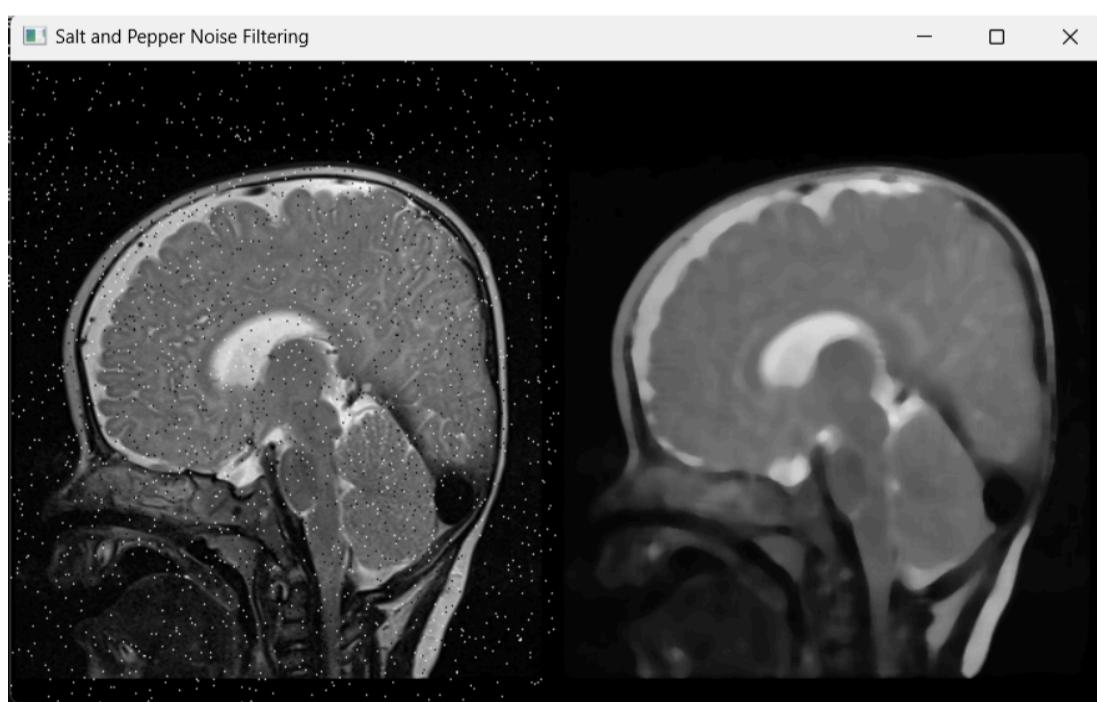
Pasted image 20241129100816.png#center

- 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 (3×3), 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 (σ)** 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 (σ)?

- 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 σ 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 3×3 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.  
|
```

Pasted image 20241202110137.png#center

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

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

Pasted image 20241202110228.png#center

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     Mat paddedImage;
5     int pad = kernelSize / 2; // pad is equal to the kernel/2
6     // Create a padded image
7     copyMakeBorder(image, paddedImage, pad, pad, pad, pad, BORDER_CONSTANT);
8
9     // 1 channel image to serve as final reference
10    Mat filterImage(image.rows, image.cols, CV_8UC1, Scalar(0));
11
12    // Loop through each pixel of the padded image, excluding the padding borders.
13    for (int r = pad; r < paddedImage.rows - pad; r++) {
14        for (int c = pad; c < paddedImage.cols - pad; c++) {
15            double sum = 0; // Change to double to match kernel type
16            // Loop through the kernel region centered at (r, c)
17            // Accumulate the sum of all pixel intensities in the kernel.
18            for (int kx = -pad; kx <= pad; kx++) { // Go through x
19                for (int ky = -pad; ky <= pad; ky++) { // Go through y
20                    // Correct data type: access kernel as double
21                    double kernelValue = kernel.at<double>(kx + pad, ky + pad); // Use CV_64F type
22                    uchar pixelValue = paddedImage.at<uchar>(r + kx, c + ky);
23
24                    // Multiply and accumulate the result / Convolve
25                    sum += static_cast<double>(pixelValue) * kernelValue;
26                }
27            }
28            // Assign the result to the output image pixels (without padding)
29            filterImage.at<uchar>(r - pad, c - pad) = static_cast<uchar>(round(sum));
30        }
31    }
32
33    // join the images in an imshow
34    Mat combinedImage;
35    hconcat(image, filterImage, combinedImage); // Combine original and filtered images side by side
36
37    imshow(imageName, combinedImage);
38    waitKey(0);
39 }
```

Referências:

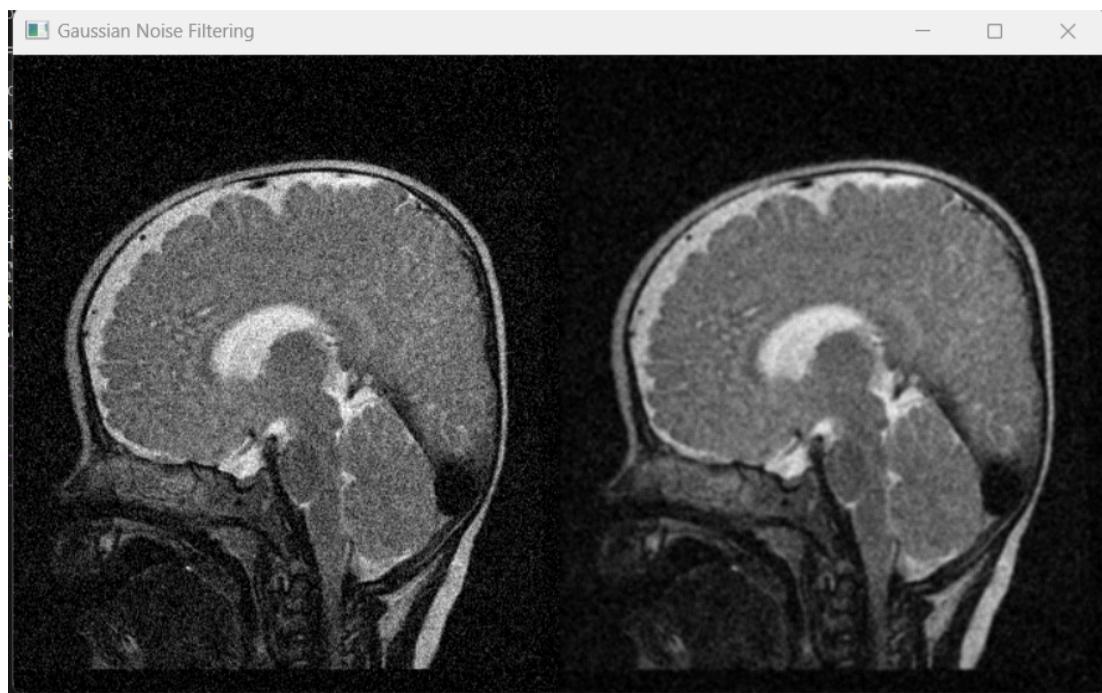
1. Width of the Bell Curve

- Sigma (σ) controls the spread or width of the Gaussian curve.
- A smaller σ :
 - 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 σ :
 - 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

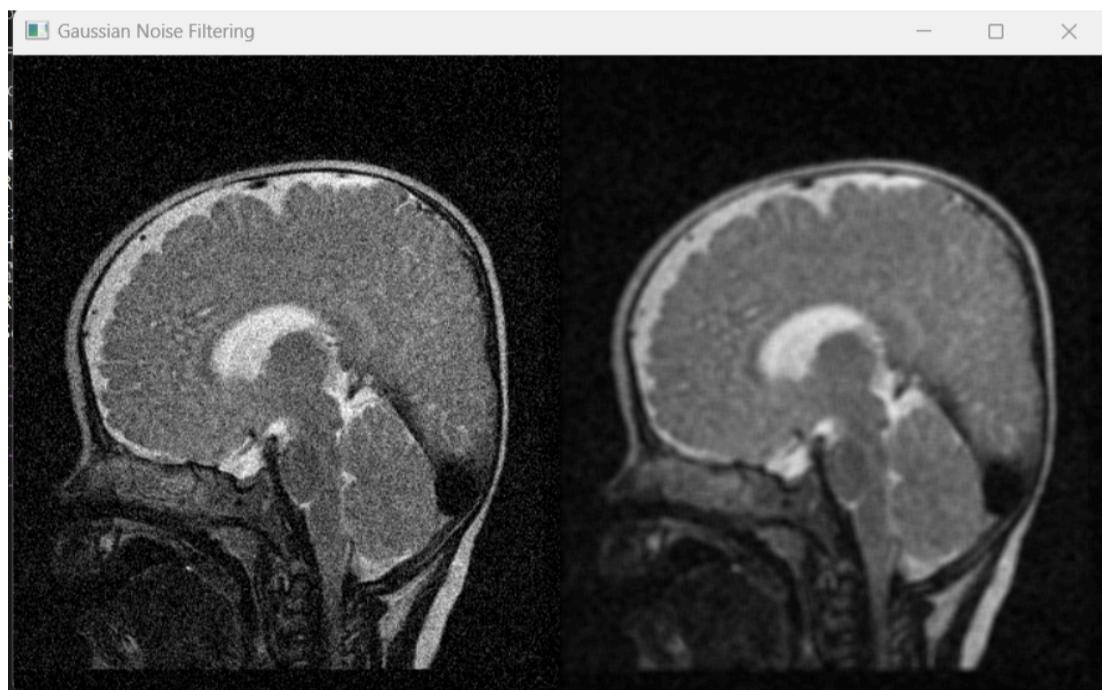
Resultados:

- Kernel 3×3 sigma 2



Pasted image 20241202111728.png#center

- Kernel 5×5 sigma 3:



Pasted image 20241202111826.png#center

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; // pad is equal to the kernel/2
6     // Create a padded image
7     copyMakeBorder(image, paddedImage, pad, pad, pad, pad, BORDER_CONSTANT);
8     // 1 channel image to serve as final reference
9     Mat filterImageHorizontal(image.rows, image.cols, CV_8UC1, Scalar(0));
10    Mat filterImageVertical(image.rows, image.cols, CV_8UC1, Scalar(0));
11    Mat filterImageMagnitude(image.rows, image.cols, CV_8UC1, Scalar(0));
12
13    float kernelx[3][3] = { {1, 0, -1},
14                           {2, 0, -2},
15                           {1, 0, -1} };
16    float kernely[3][3] = { {1, 2, 1},
17                           {0, 0, 0},
18                           {-1, -2, -1} };
19
20    for (int r = pad; r < paddedImage.rows - pad; r++) {
21        for (int c = pad; c < paddedImage.cols - pad; c++) {
22            int sum_x = 0, sum_y = 0; // sumn for both axis
23            for (int kx = -pad; kx <= pad; kx++) { // go through x
24                for (int ky = -pad; ky <= pad; ky++) { // go through y
25                    int pixel_value = paddedImage.at<uchar>(r + kx, c + ky);
26                    sum_x += pixel_value * kernelx[kx + pad][ky + pad]; // Convolution with Sobel X
27                    sum_y += pixel_value * kernely[kx + pad][ky + pad]; // Convolution with Sobel Y
28                }
29            }
30            // Sobel x results / horizontal
31            filterImageHorizontal.at<uchar>(r - pad, c - pad) = static_cast<int>(round(float(sum_x)));
32            // Sobel x results / horizontal
33            filterImageVertical.at<uchar>(r - pad, c - pad) = static_cast<int>(round(float(sum_y)));
34            // Combine Sobel X and Y results
35            int magnitude = sqrt(sum_x * sum_x + sum_y * sum_y);
36            filterImageMagnitude.at<uchar>(r - pad, c - pad) = static_cast<int>(round(magnitude));
37        }
38    }
39    // low pass filter
40    Mat lowPassSobelHorizontal = lowpassFilter(image, filterImageHorizontal, pad);
41    Mat lowPassSobelVertical = lowpassFilter(image, filterImageVertical, pad);
42    Mat lowPassSobelMagnitude = lowpassFilter(image, filterImageMagnitude, pad);
43    // Combine images for visualization
44    Mat combinedImage;
45    Mat combinedImage2;
46    Mat combinedImage3;
47    hconcat(image, lowPassSobelHorizontal, combinedImage);
48    hconcat(combinedImage, lowPassSobelVertical, combinedImage2);
49    hconcat(combinedImage2, lowPassSobelMagnitude, combinedImage3);
50    // Display the final combined image
51    imshow(imageName, combinedImage3);
52    waitKey(0);
53 }
```

lowpassfilter

```
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            // Average the kernel values
18            lowPass.at<uchar>(r - pad, c - pad) = saturate_cast<uchar>(sum / (3 * 3));
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

 Cric Lienco

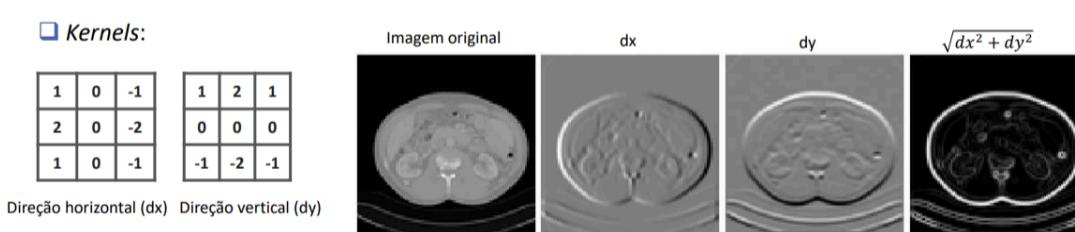
answered Jul 24, 2013 at 9:18

 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

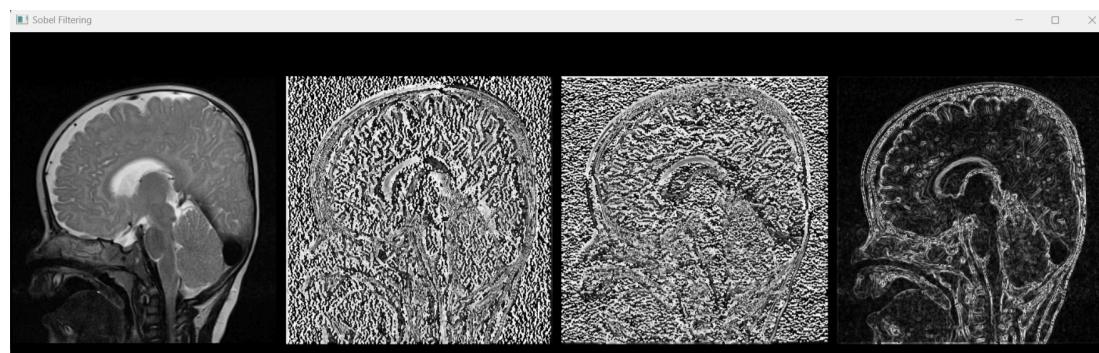
Pasted image 20241202142414.png#center



Pasted image 20241202115858.png#center

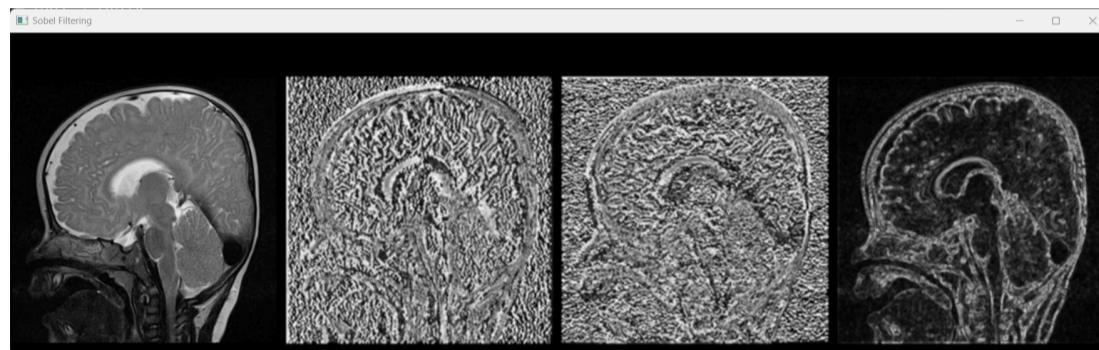
Resultados:

- Sem filtro passa baixo:



Pasted image 20241202115820.png#center

- Com filtro passa baixo de média:



Pasted image 20241202142221.png#center

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 the kernel/2
6     // Create a padded image
7     copyMakeBorder(image, paddedImage, pad, pad, pad, pad, BORDER_CONSTANT);
8     // 1 channel image to serve as final reference
9     Mat filterLaplace(image.rows, image.cols, CV_8UC1, Scalar(0));
10
11    float kernel[3][3] = { {0, 1, -1},
12                           {1, -4, 1},
13                           {0, 1, 0} };
14
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            for (int kx = -pad; kx <= pad; kx++) { // go through x
19                for (int ky = -pad; ky <= pad; ky++) { // go through y
20                    int pixel_value = paddedImage.at<uchar>(r + kx, c + ky);
21                    sum += pixel_value * kernel[kx + pad][ky + pad]; // Convolution with Laplace
22                }
23            }
24            // Laplace result
25            filterLaplace.at<uchar>(r - pad, c - pad) = static_cast<int>(round(float(sum)));
26        }
27    }
28    // low pass filter
29    Mat lowPassLaplace = lowpassFilter(image, filterLaplace, pad);
30
31    // Combine images for visualization
32    Mat combinedImage;
33    Mat combinedImage2;
34    hconcat(image, filterLaplace, combinedImage);
35    hconcat(combinedImage, lowPassLaplace, combinedImage2);
36
37    // Display the final combined image
38    imshow(imageName, combinedImage2);
39    waitKey(0);
40 }
```

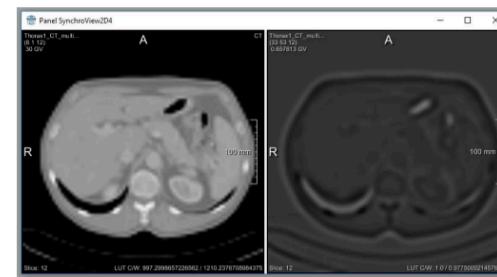
lowpass

```
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:

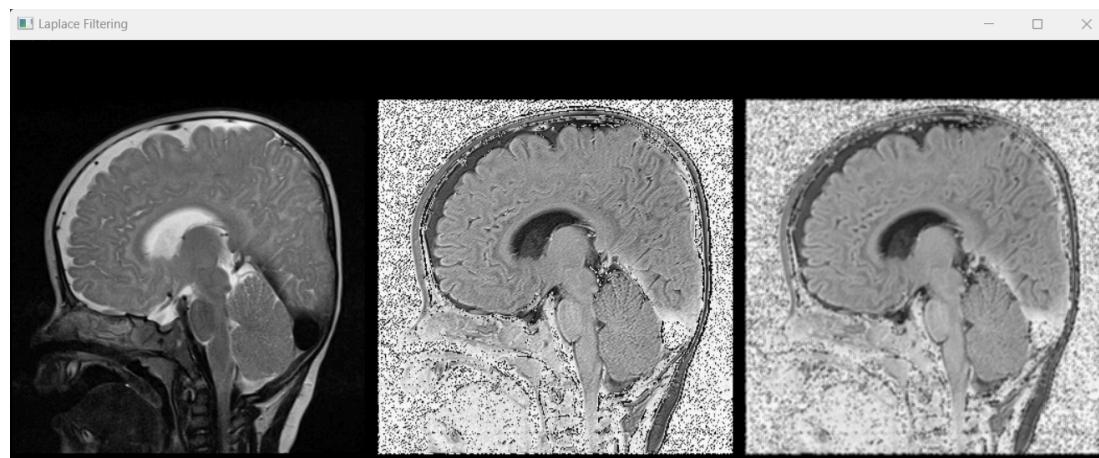
- Calcula a segunda derivada da imagem
- Realçar transições rápidas de intensidade
- Deteção de contornos
- Muitas vezes precedido por um filtro passa-baixo
- Kernel:

0	1	0
1	-4	1
0	1	0



Pasted image 20241202142730.png#center

Resultados:



Pasted image 20241202143328.png#center