

Universidade Estadual Paulista - UNESP  
Programa de Pós-Graduação em Ciência da  
Computação  
Processamento de Imagens Digitais  
Professor Dr. Leandro Alves Neves

## **Relatório exercícios**

Giovanna Carreira Marinho  
Guilherme Francisco de Andrade Campos  
Julia Rodrigues Gubolin  
Thales Ricardo de Souza Lopes

# Conteúdo

<b>1 Aspectos técnicos sobre a implementação</b>	<b>2</b>
<b>2 Exercícios aula 4 - Ruído</b>	<b>2</b>
2.1 Exercício 5)	2
2.2 Exercício 6)	5
<b>3 Exercícios aula 6 - Filtragem</b>	<b>6</b>
3.1 Exercício 4)	6
3.2 Exercício 6)	7
<b>4 Exercícios aula 7 - Modelo de Cores e Fatiamento</b>	<b>31</b>
4.1 Exercício 3)	31
4.2 Exercício 5)	34
<b>5 Exercícios aula 8 - Filtragem no domínio da frequência</b>	<b>35</b>
5.1 Exercício 4)	35
<b>6 Exercícios aula 9 - Segmentação de imagens</b>	<b>40</b>
6.1 Exercício 2)	40
<b>7 Exercícios aula 10 - Morfologia Matemática</b>	<b>43</b>
7.1 Exercício 4)	43
<b>8 Exercícios aula 11 - Representação, descrição e análise de textura</b>	<b>45</b>
8.1 Exercício 1)	45

# 1 Aspectos técnicos sobre a implementação

As implementações dos algoritmos foram feitas na linguagem Python, utilizando o ambiente [Google Colab](#), que permite a escrita e execução de códigos nessa linguagem no navegador, sem nenhuma configuração prévia necessária, facilitando o compartilhamento da implementação.

Para cada implementação (organizadas em aulas) um arquivo com extensão `.ipynb` é criado pela ferramenta. Arquivos podem ser abertos pelo caminho: "Arquivo» "Abrir notebook» basta selecionar o respectivo arquivo a ser aberto (para carregar arquivos locais, selecione a aba "Upload" para escolher o arquivo).

O Google Colab consiste em um ambiente interativo (notebook), que permite escrever e executar códigos em estruturas de blocos/células. Para executar o código de uma célula, clique nela e depois pressione o botão Play à esquerda do código ou use o atalho do teclado "Command/Ctrl+Enter". Caso seja necessário executar todo o arquivo, clique na aba "Ambiente de execução» "Executar tudo".

Além da linguagem Python, foram utilizados pacotes disponíveis na linguagem para manipulação de imagens, matrizes, etc. Entre eles, destacam-se `skimage`, `numpy`, `matplotlib` e `OpenCV`.

## 2 Exercícios aula 4 - Ruído

### 2.1 Exercício 5)

5) Considere as imagens obtidas no exercício 3 e aplique a correção gama com  $c=1$  e  $\gamma(0.04; 0.4; 1; 2, 5; 10; e 25)$ . Visualmente, esse tipo de realce permitiu melhorar a qualidade de cada imagem com ruído? Se houve algum resultado positivo, indique o valor de  $\gamma$ , o tipo de ruído e a imagem.

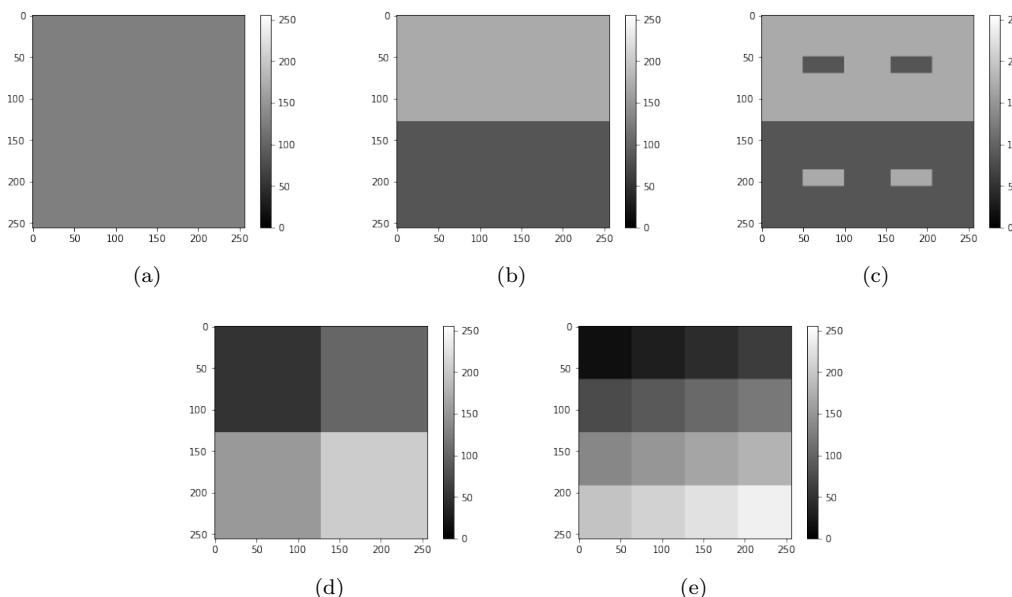


Figura 1: Imagens geradas pelo programa como resposta ao exercício 3.

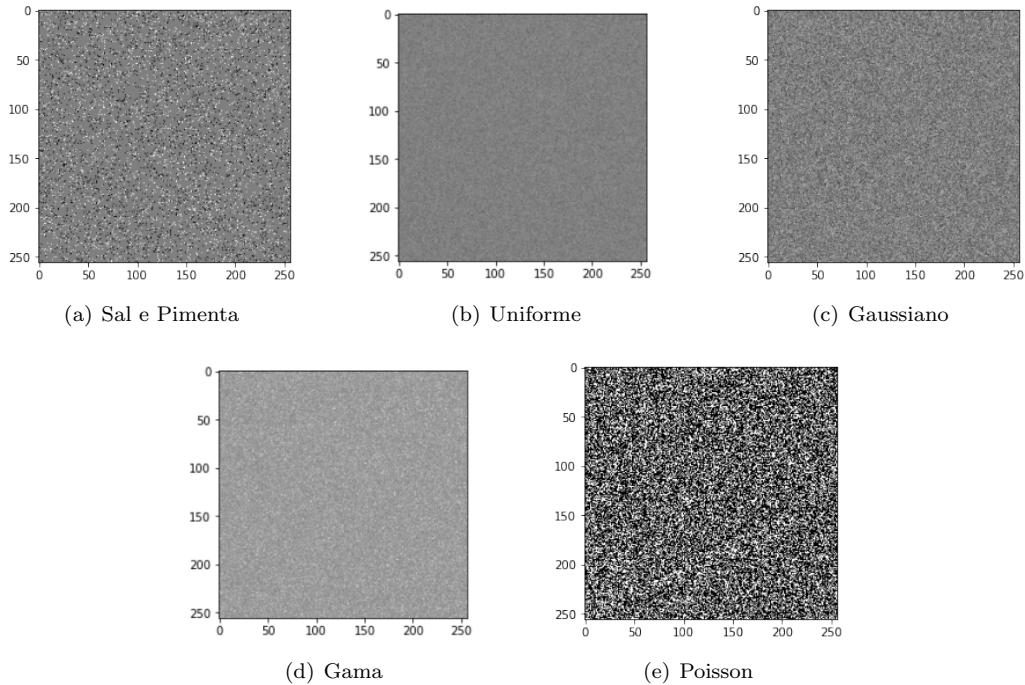


Figura 2: Imagem (a) com os ruídos: sal e pimenta, uniforme, gaussiano, gama e poisson.

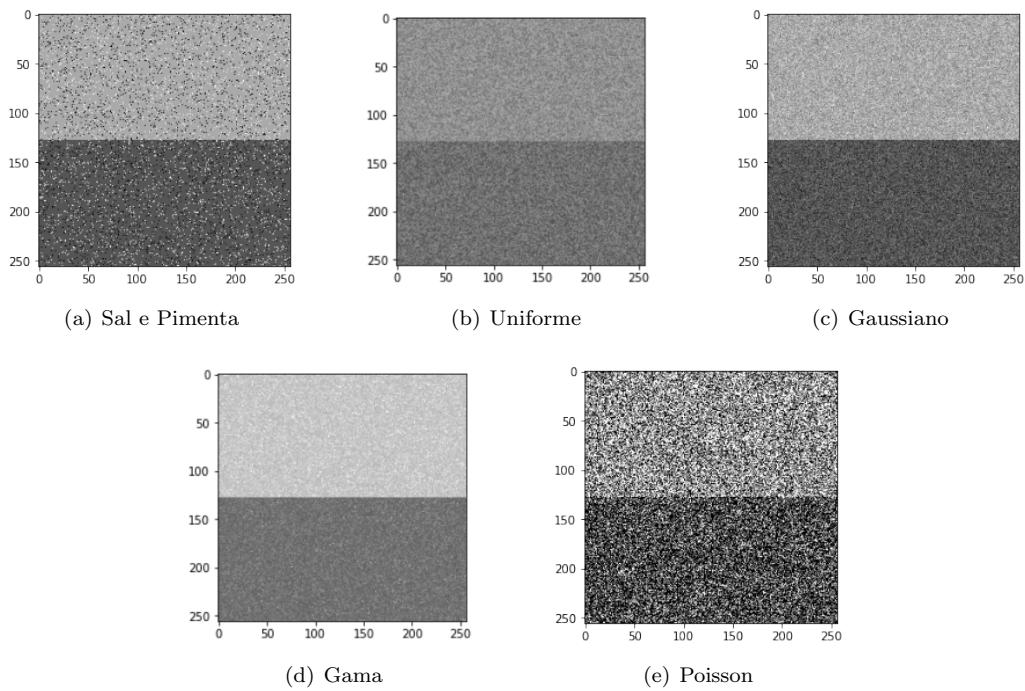


Figura 3: Imagem (b) com os ruídos: sal e pimenta, uniforme, gaussiano, gama e poisson.

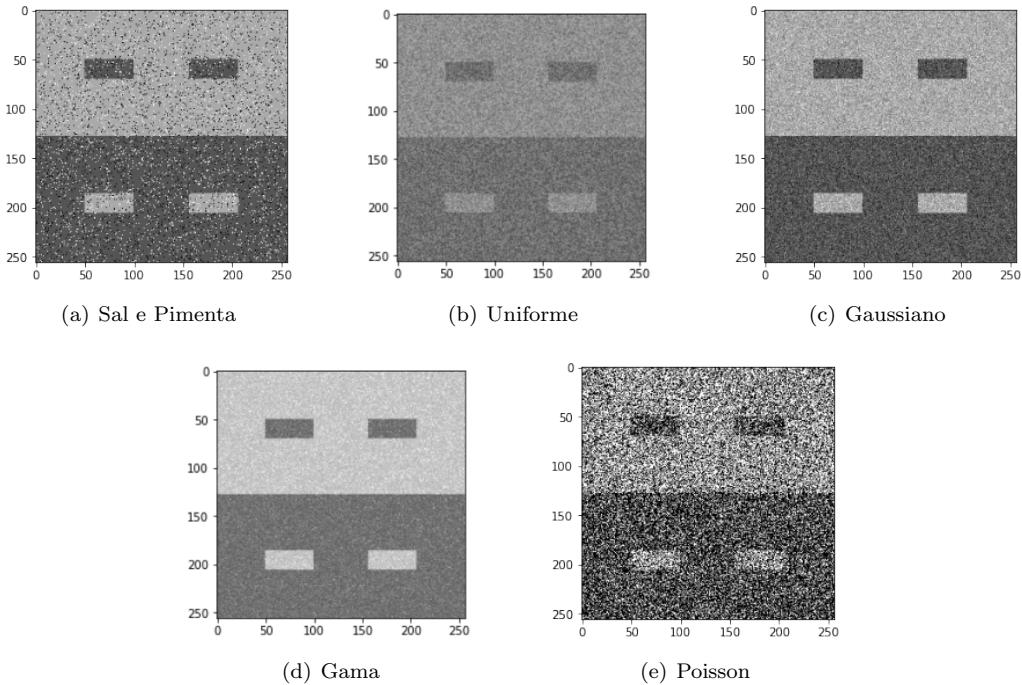


Figura 4: Imagem (c) com os ruídos: sal e pimenta, uniforme, gaussiano, gama e poisson.

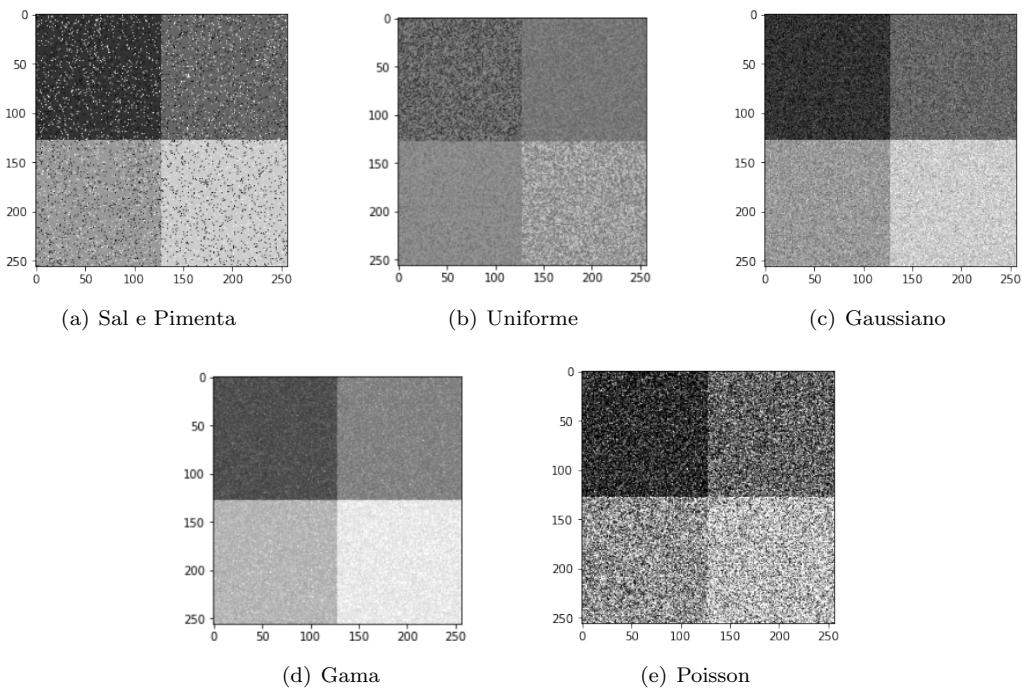


Figura 5: Imagem (d) com os ruídos: sal e pimenta, uniforme, gaussiano, gama e poisson.

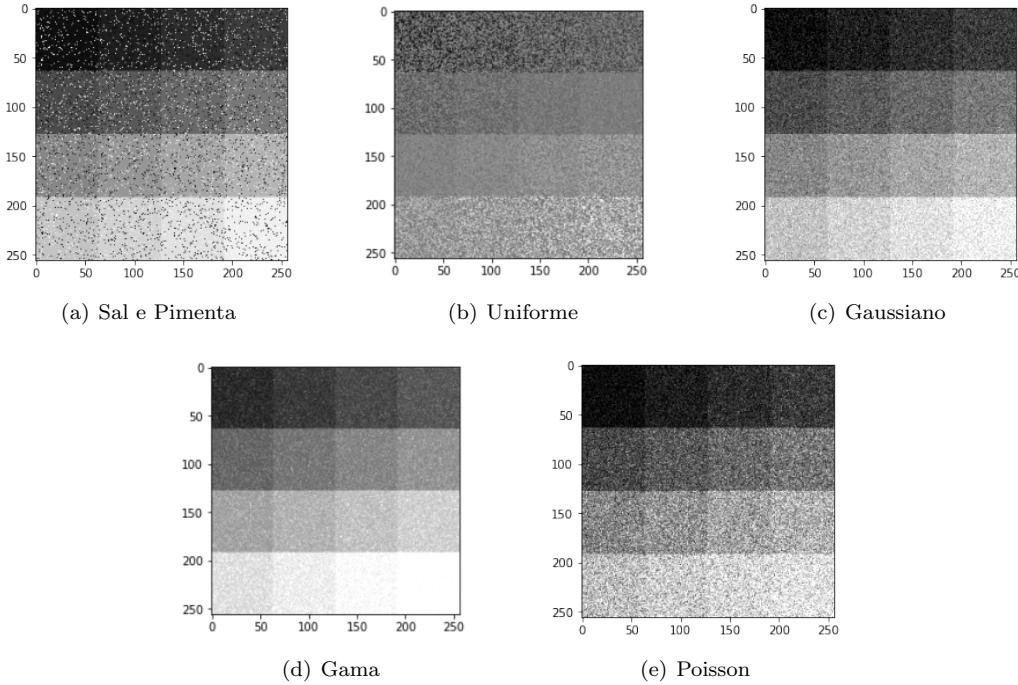


Figura 6: Imagem (e) com os ruídos: sal e pimenta, uniforme, gaussiano, gama e poisson.

**Resposta:** Nenhuma das imagens com ruído se mostrou visualmente melhor após a aplicação da correção gama. Por mais que a imagem ficasse mais clara ou mais escura de forma geral, o ruído ainda se manteve presente da mesma forma. Um exemplo pode ser observado na Figura 7, onde foi aplicado o ruído aditivo gama sobre a imagem (d) e depois a correção gama com  $\gamma = 0.4$  e  $c = 1$ . Mesmo com a aplicação da técnica, é possível visualizar claramente o ruído na última imagem. O código para a aplicação da correção gama segue abaixo:

```
# Função que recebe uma imagem, o valor de gama e c
# e retorna uma imagem com a correção gama aplicada
def gamma_correction(f, gamma = 1, c = 1):
    return (c*((f/255)**gamma))*255
```

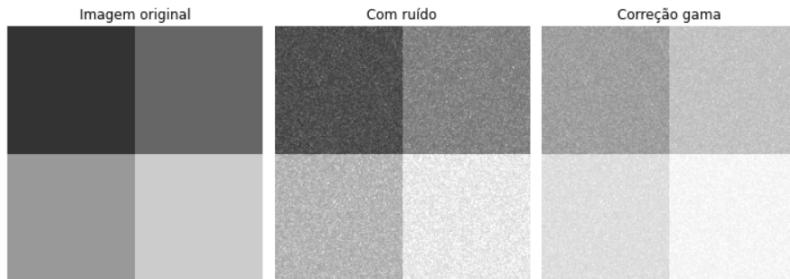


Figura 7: Imagem (a) original, seguida da aplicação do ruído gama e correção gama com  $\gamma = 0.4$

## 2.2 Exercício 6)

- 6) Após aplicar a correção gama com  $c=1$  e  $\gamma(0.04; 0.4; 1; 2, 5; 10; \text{ e } 25)$  nas imagens com ruídos, calcule as métricas indicadas no exercício 4 e avalie se é possível comprovar quantitativamente as observações empíricas indicadas no exercício 5. Por fim, responda se esse tipo de técnica (realce) é útil para corrigir as imagens com ruídos. Justifique sua resposta.

**Resposta:** A partir das métricas de avaliação de erro é possível comprovar que a correção gama não foi eficiente em eliminar o ruídos aditivos das imagens, visto que os erros calculados mostram que as imagens possuem um grau de diferença alto, como mostra as medidas de erro na Tabela 1, onde são mostrados os erros das imagens original e após a correção gama do exemplo da Figura 7.

Tabela 1: Erros medidos para a imagem (d)

Métrica	Erro
Erro máximo	173.15
Erro médio absoluto	78.79
Erro médio quadrático	6941.47
Raíz do erro médio quadrático	83.32
Erro médio quadrático normalizado	0.045
Relação sinal-ruído de pico	9.71
Covariância	1816.56
Coeficiente de Jaccard	0.0

### 3 Exercícios aula 6 - Filtragem

#### 3.1 Exercício 4)

4) Crie um programa para gerar discretamente máscaras 3x3, 5x5 e 7x7 representativas de filtros gaussianos. Use os coeficientes da expansão binomial de Newton. Para cada máscara, calcule o valor de  $\sigma$

**Resposta:** As máscaras foram geradas utilizando as linhas três, cinco e sete do triângulo de Pascal, para gerar as matrizes 3x3, 5x5 e 7x7, respectivamente. O código para a geração do filtro gaussiano segue abaixo:

```
# Função que recebe o valor da linha do triângulo de Pascal
# que será utilizado para a criação da máscara.

def get_pascal_Triangle(size = 3):
    if size == 3:
        pascal_line = np.array([[1], [2], [1]], dtype=np.uint32)
    elif size == 5:
        pascal_line = np.array([[1],[4], [6], [4], [1]], dtype=np.uint32)
    elif size == 7:
        pascal_line = np.array([[1], [6], [15], [20], [15], [6], [1]], dtype=np.uint32)

    # Calcula o desvio padrão para aquela linha
    delta = np.sqrt((size-1)/2)

    # Retorna os vetor com os valores da linha de pascal
    # e o desvio padrão
    return pascal_line, delta

# Função que recebe o tamanho do kernel e gera a máscara do filtro gaussiano
def get_Gaussian_Filter(size = 3):
    line, delta = get_pascal_Triangle(size)

    # Multiplica o vetor da linha de Pascal escolhida
    # com a transposta dela, para gerar a matriz NxN
    gaussian_filter = np.matmul(line, np.transpose(line))

    print(f"Desvio padrão: {delta}")
    print(f"\nFiltro Gaussiano: \n {gaussian_filter}")
```

```
return gaussian_filter.astype(int)
```

$$M_{3x3} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

$$M_{5x5} = \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

$$M_{7x7} = \begin{bmatrix} 1 & 6 & 15 & 20 & 15 & 6 & 1 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 20 & 120 & 300 & 400 & 300 & 120 & 20 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{bmatrix}$$

Também foi calculado os valores de desvio padrão para cada máscara. Para a máscara  $M_{3x3}$  o desvio padrão é de  $\sigma = 1$ , para  $M_{5x5}$  esse valor foi de  $\sigma = 1.41$ , e por fim, para  $M_{7x7}$  o valor é de  $\sigma = 1.73$

### 3.2 Exercício 6)

6) Aplique sobre cada imagem indicada a seguir os ruídos aditivos: sal e pimenta; uniforme e gaussiano. As distribuições devem ser fornecidas pelo usuário. Aplique os filtros apresentados abaixo.

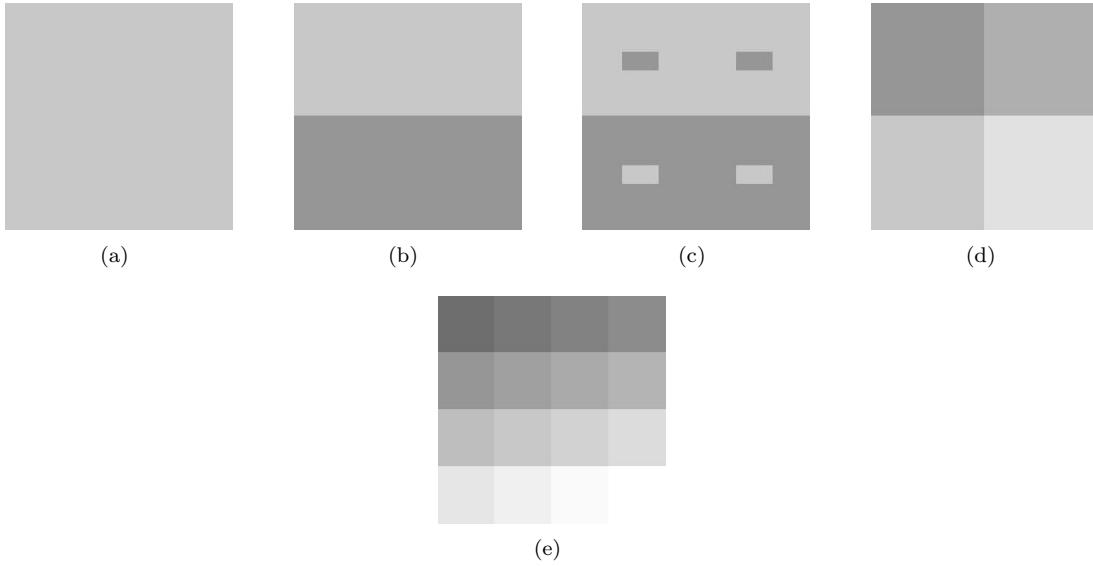


Figura 8: Imagens do enunciado.

- a) Suavização da imagem (Média, Mediana, Gaussiano, Mínimo, Máximo e da Moda), com janelas de 3x3 e 5x5;
- b) Filtro Passa-Alta com as máscaras:

$$h_1 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad h_2 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Figura 9: Imagem do enunciado.

- c) Considerando  $i$  como sendo cada imagem dada como entrada, determine qual filtro indicou o melhor resultado visual  $\hat{i}$ . Em seguida, use ao menos três métricas para avaliar a qualidade de  $\hat{i}$  e confirmar sua hipótese.

**Resposta:** Primeiramente, foram carregadas as imagens e aplicados os ruídos Sal e Pimenta (com probabilidade  $p = 0.05$ ), Gaussiano (com média  $mean = 0$  e variância  $var = 0.01$ ) e Uniforme (com intervalo definido de  $a = 100$  a  $b = 150$ ). O código para a aplicação dos ruídos segue abaixo:

```
# Carrega a imagem do caminho especificado
img = cv.imread("image.jpg")
img = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

# Aplica o ruído sal e pimenta
probability = 0.05
salt_noised_img = skimage.util.random_noise(img, mode='s&p', amount=probability)*255

...
# Aplica o ruído gaussiano
gaussian_noised_img = skimage.util.random_noise(img, mode='gaussian', mean=0, var=0.01)*255

...
# Aplica o ruído uniforme
uniform_noised_img = apply_uniform_noise(img, 100, 150)
```

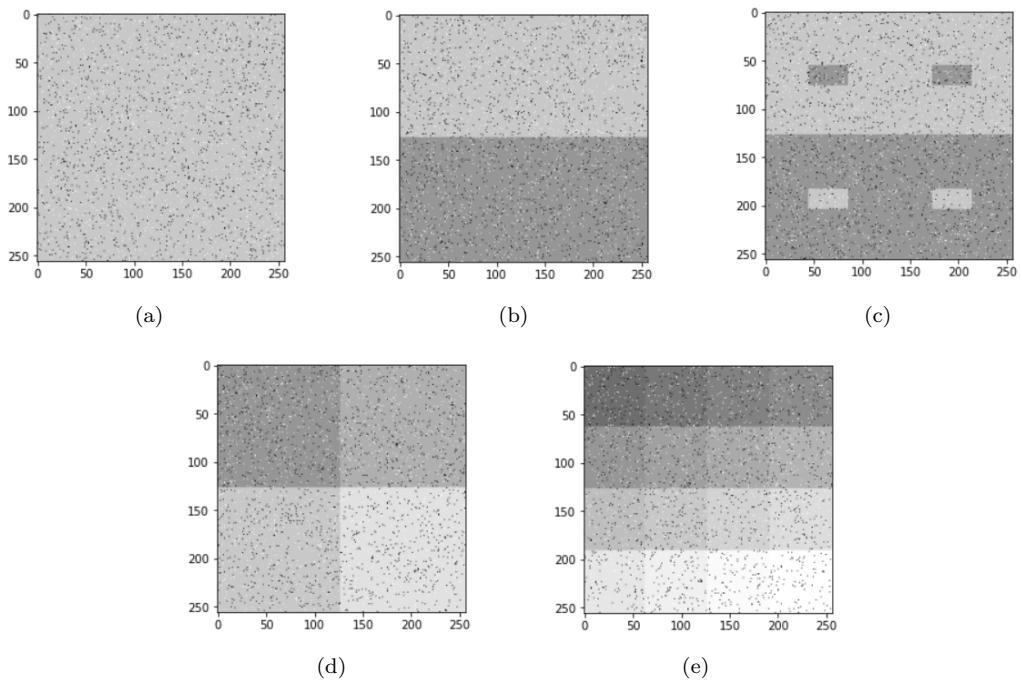


Figura 10: Imagens com o ruído sal e pimenta.

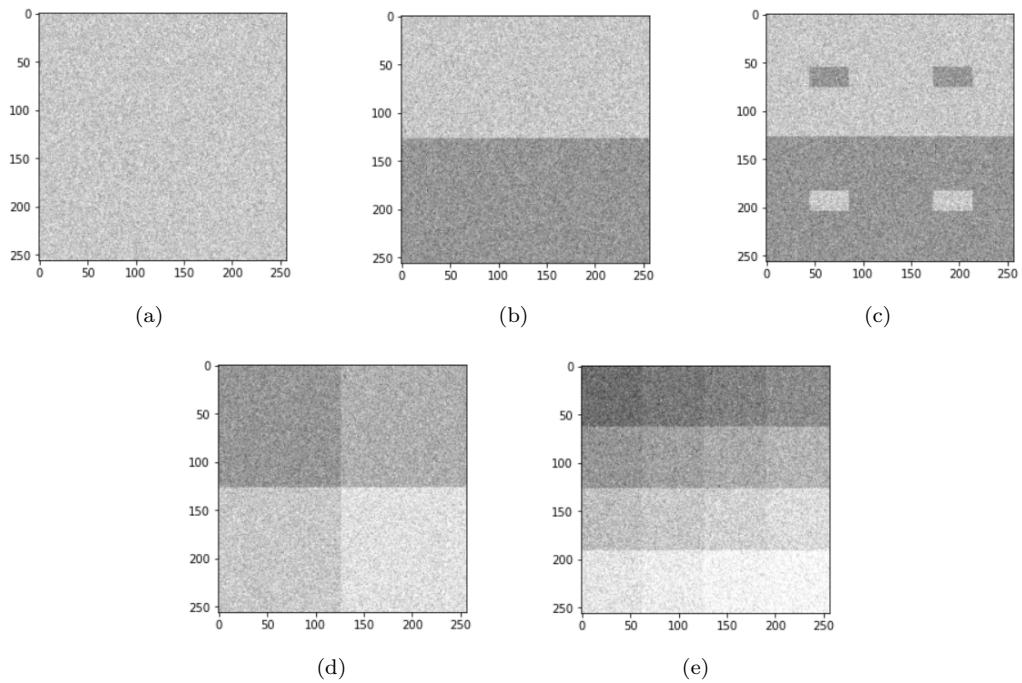


Figura 11: Imagens com o ruído gaussiano.

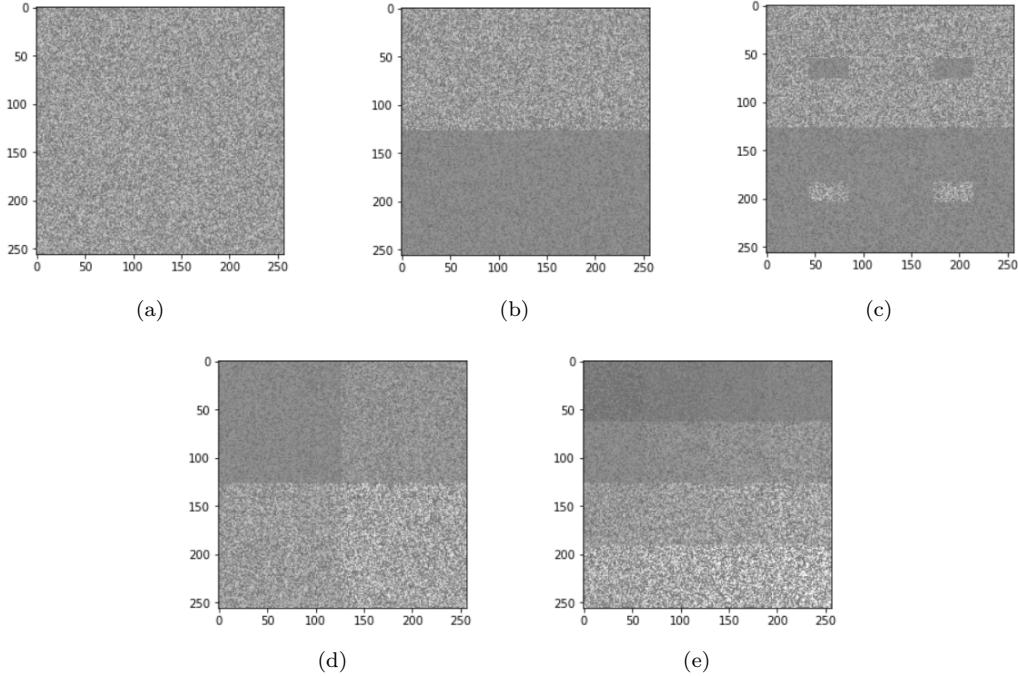


Figura 12: Imagens com o ruído uniforme.

a) Após a aplicação dos ruídos, foram utilizadas os filtros de Média, Mediana, Gaussiano, Mínimo, Máximo e da Moda, com kernel de tamanho  $3 \times 3$  e  $5 \times 5$  para cada filtro. O código para a aplicação dos filtros segue abaixo:

```
# Aplicação do filtro da média com máscara 3x3
filter_img = filtro_média(noised_img, 3)

# Aplicação do filtro da média com máscara 5x5
filter_img = filtro_média(noised_img, 5)

# Aplicação do filtro da mediana com máscara 3x3
filter_img = filtro_mediana(noised_img, 3)

# Aplicação do filtro da mediana com máscara 5x5
filter_img = filtro_mediana(noised_img, 5)

# Aplicação do filtro gaussiano com máscara 3x3
filter_img = filtro_gaussiano(noised_img, 3)

# Aplicação do filtro gaussiano com máscara 5x5
filter_img = filtro_gaussiano(noised_img, 5)

# Aplicação do filtro mínimo com máscara 3x3
filter_img = filtro_minimo(noised_img, 3)

# Aplicação do filtro mínimo com máscara 5x5
filter_img = filtro_minimo(noised_img, 5)

# Aplicação do filtro máximo com máscara 3x3
filter_img = filtro_maximo(noised_img, 3)

# Aplicação do filtro máximo com máscara 5x5
filter_img = filtro_maximo(noised_img, 5)
```

```

# Aplicação do filtro da moda com máscara 3x3
filter_img = filtro_moda(noised_img, 3)

# Aplicação do filtro da moda com máscara 5x5
filter_img = filtro_moda(noised_img, 5)

```

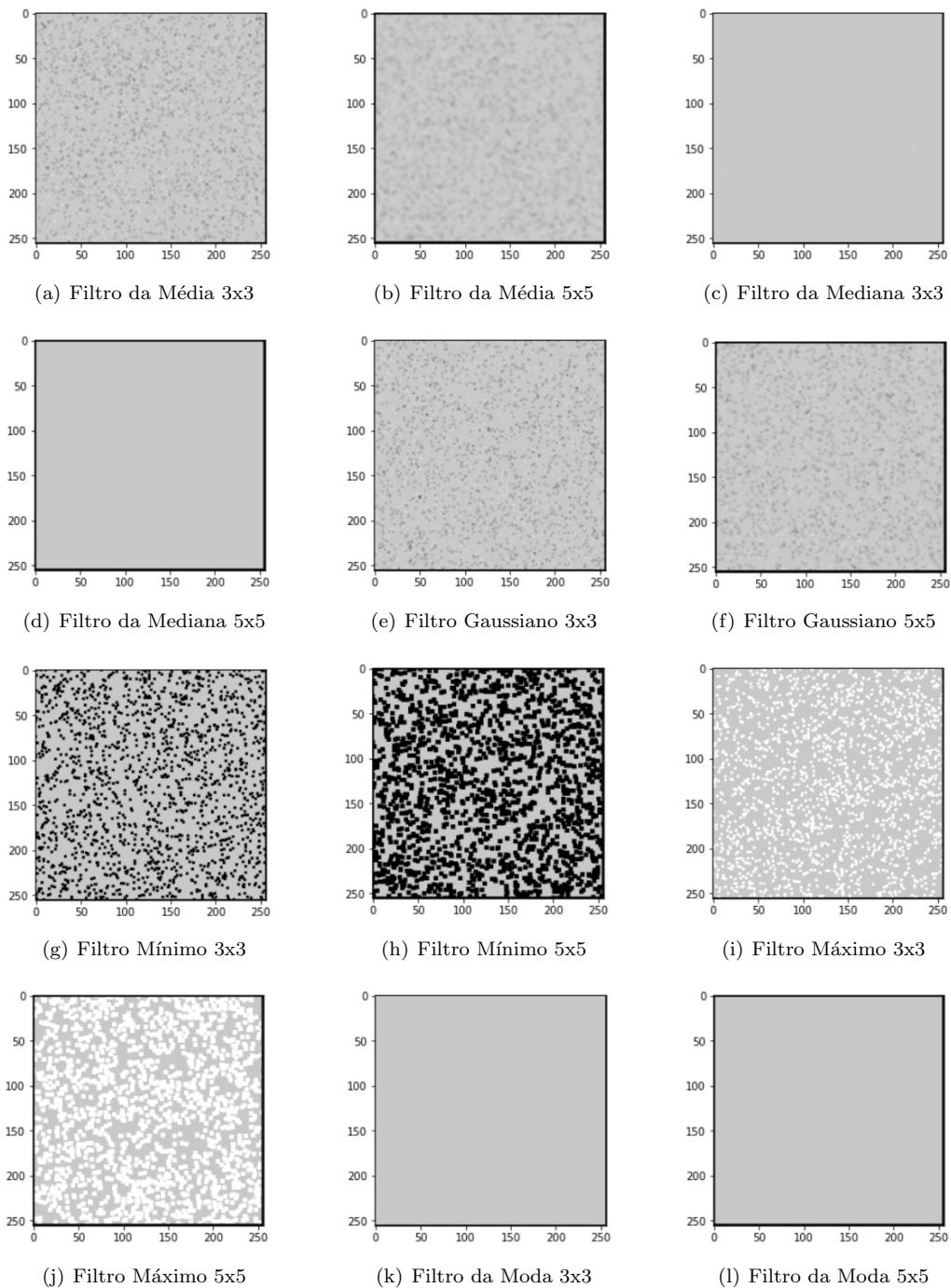


Figura 13: Imagem (a) com ruído sal e pimenta suavizada pelos filtros.

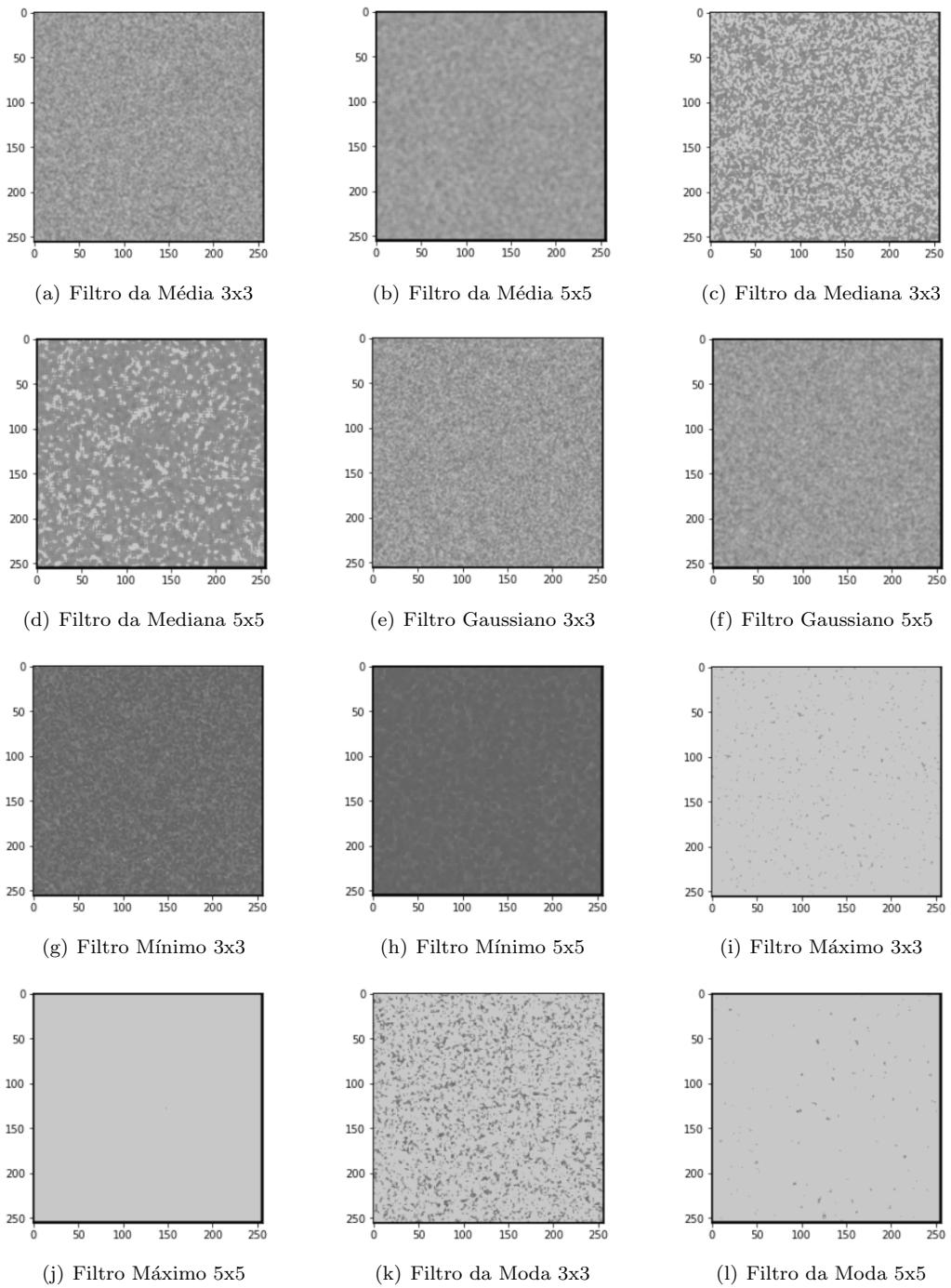


Figura 14: Imagem (a) com ruído uniforme suavizada pelos filtros.

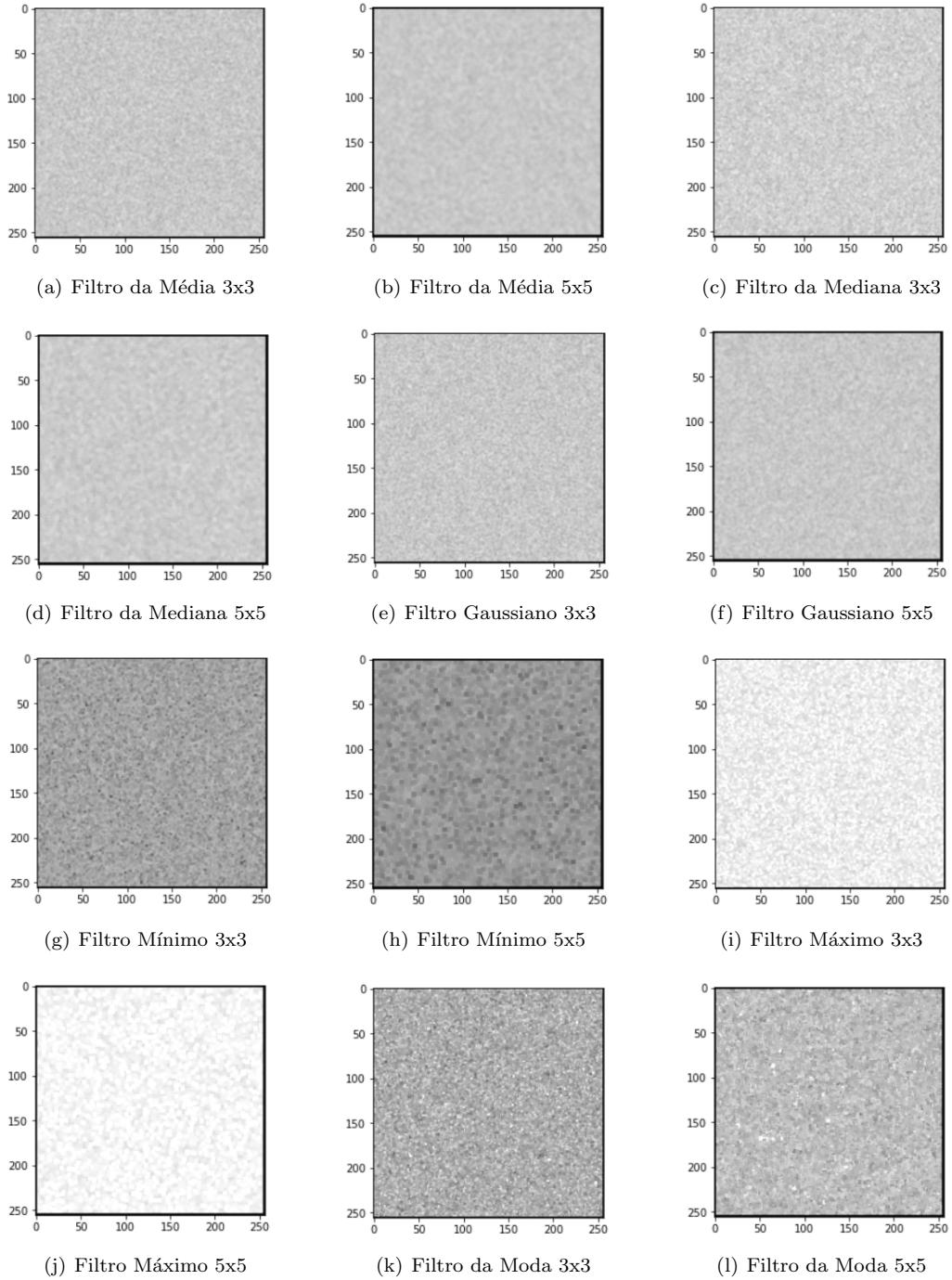


Figura 15: Imagem (a) com ruído gaussiano suavizada pelos filtros.

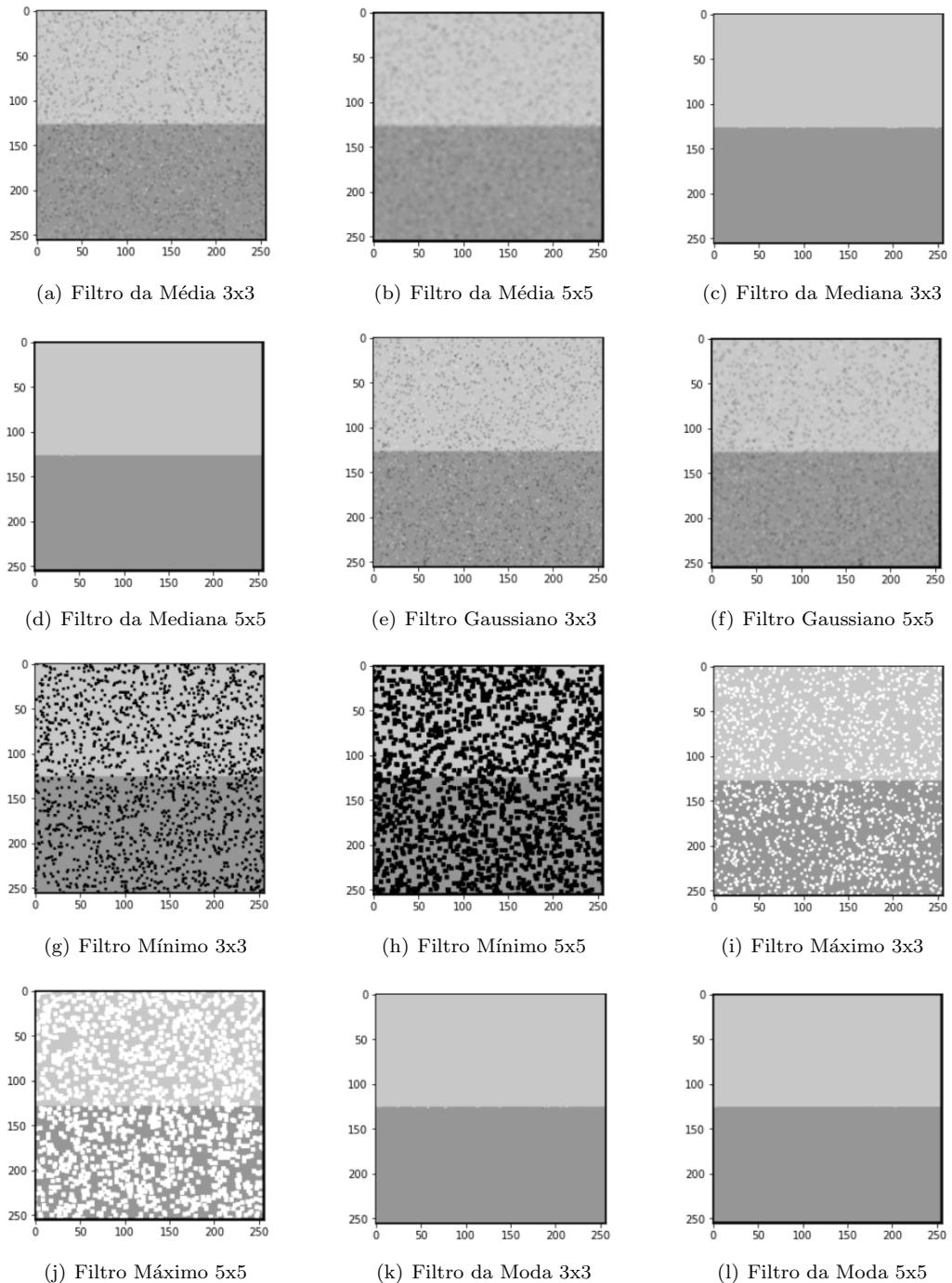


Figura 16: Imagem (b) com ruído sal e pimenta suavizada pelos filtros.

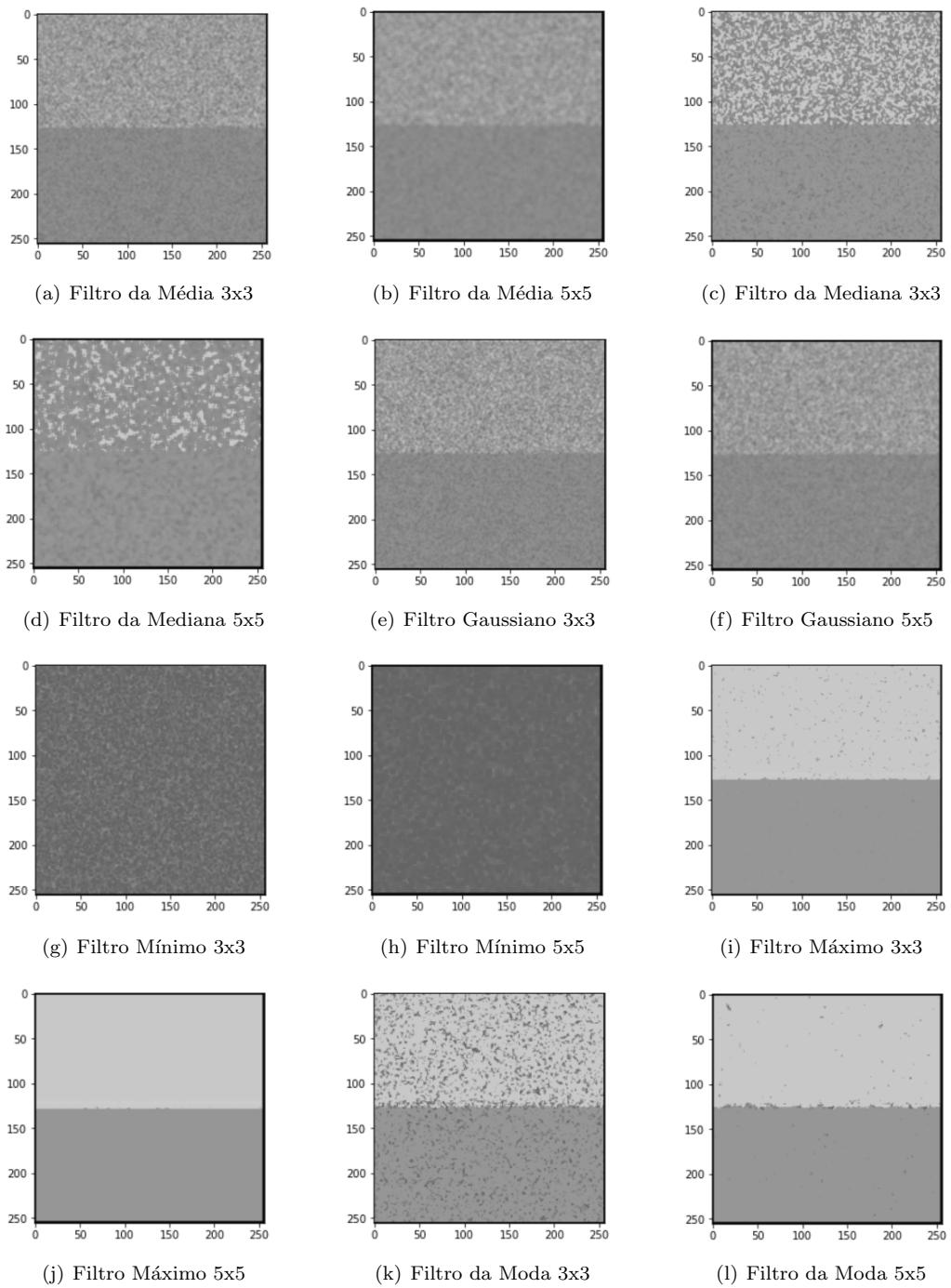


Figura 17: Imagem (b) com ruído uniforme suavizada pelos filtros.

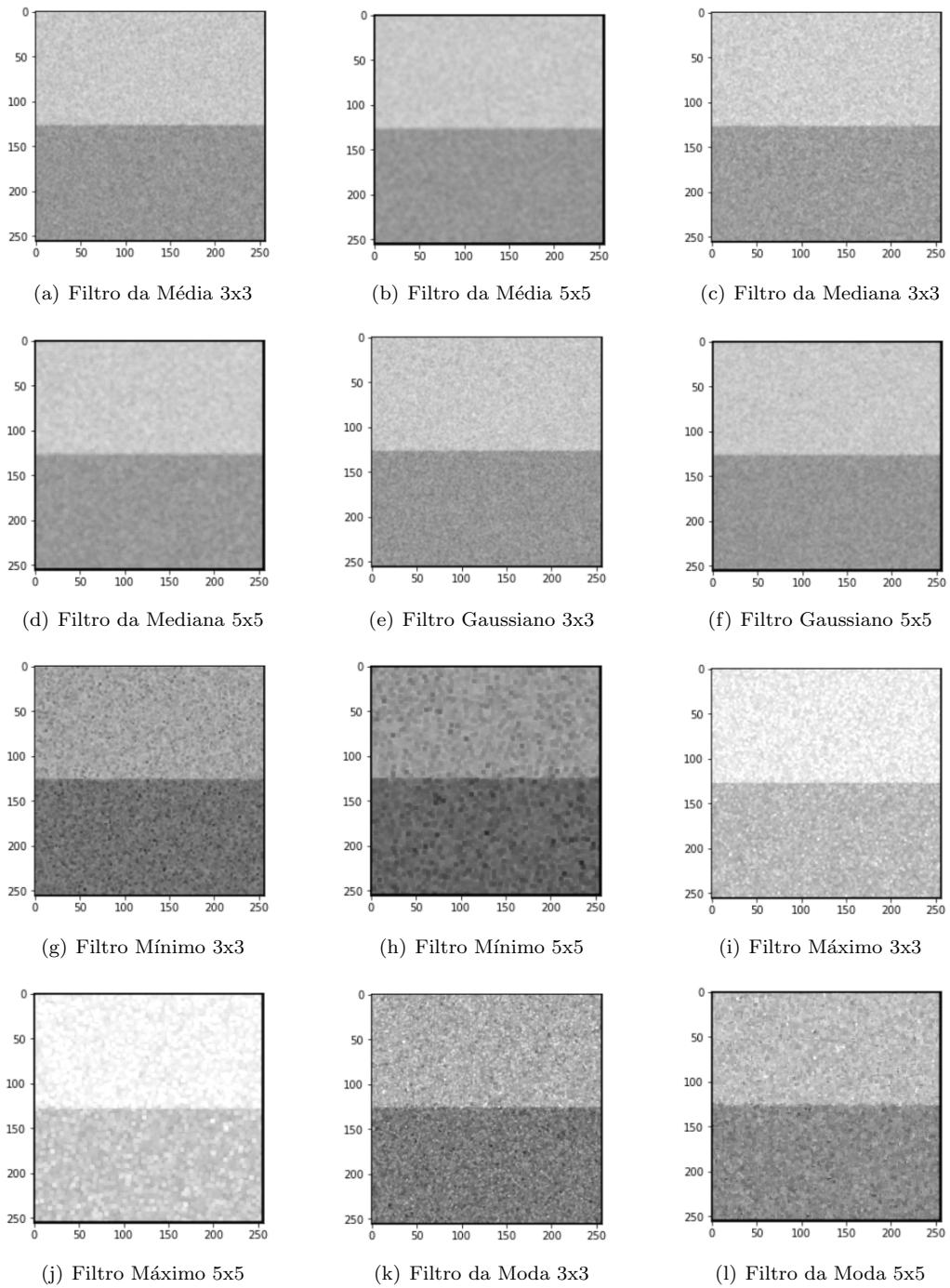


Figura 18: Imagem (b) com ruído gaussiano suavizada pelos filtros.

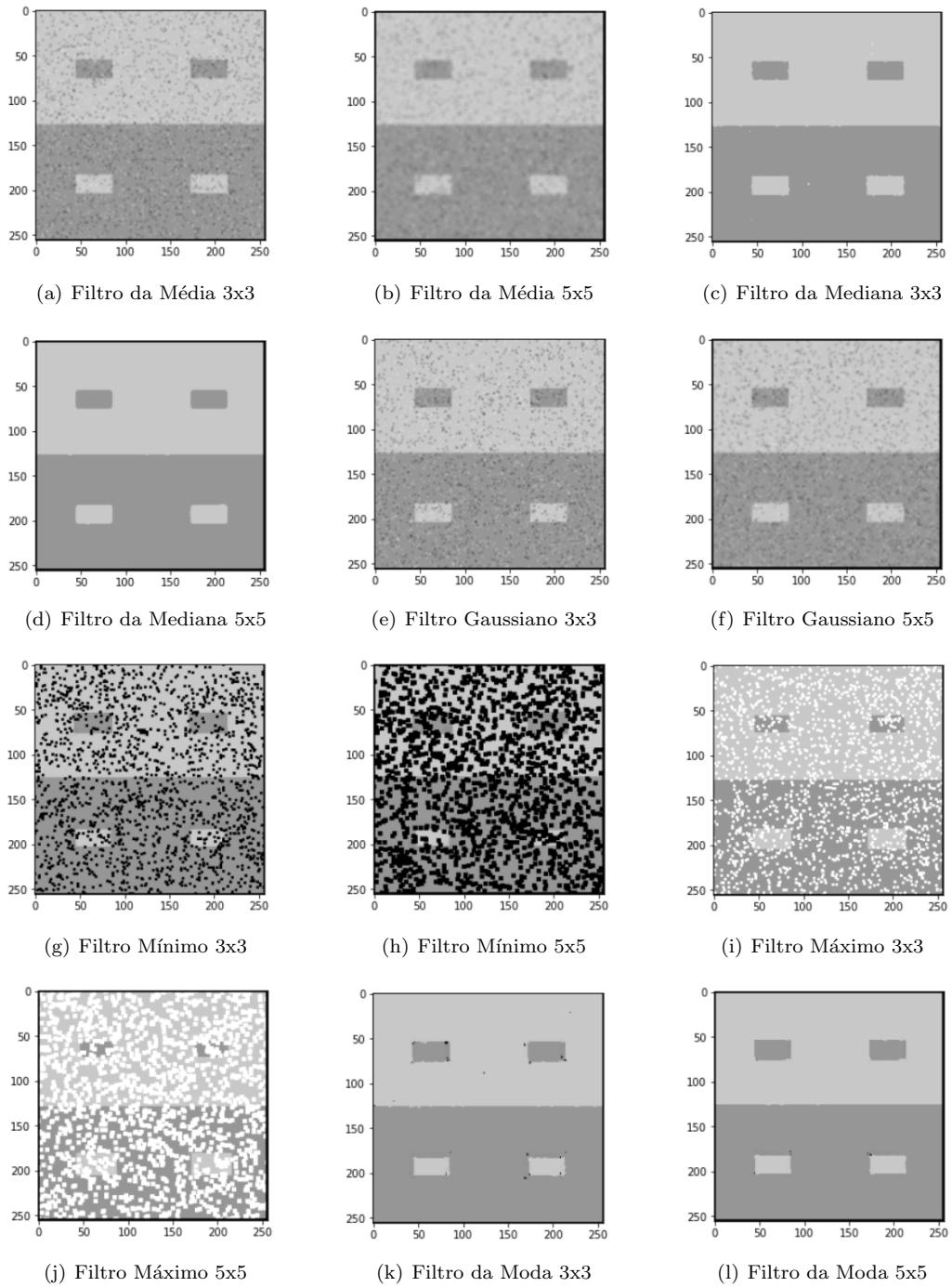


Figura 19: Imagem (c) com ruído sal e pimenta suavizada pelos filtros.

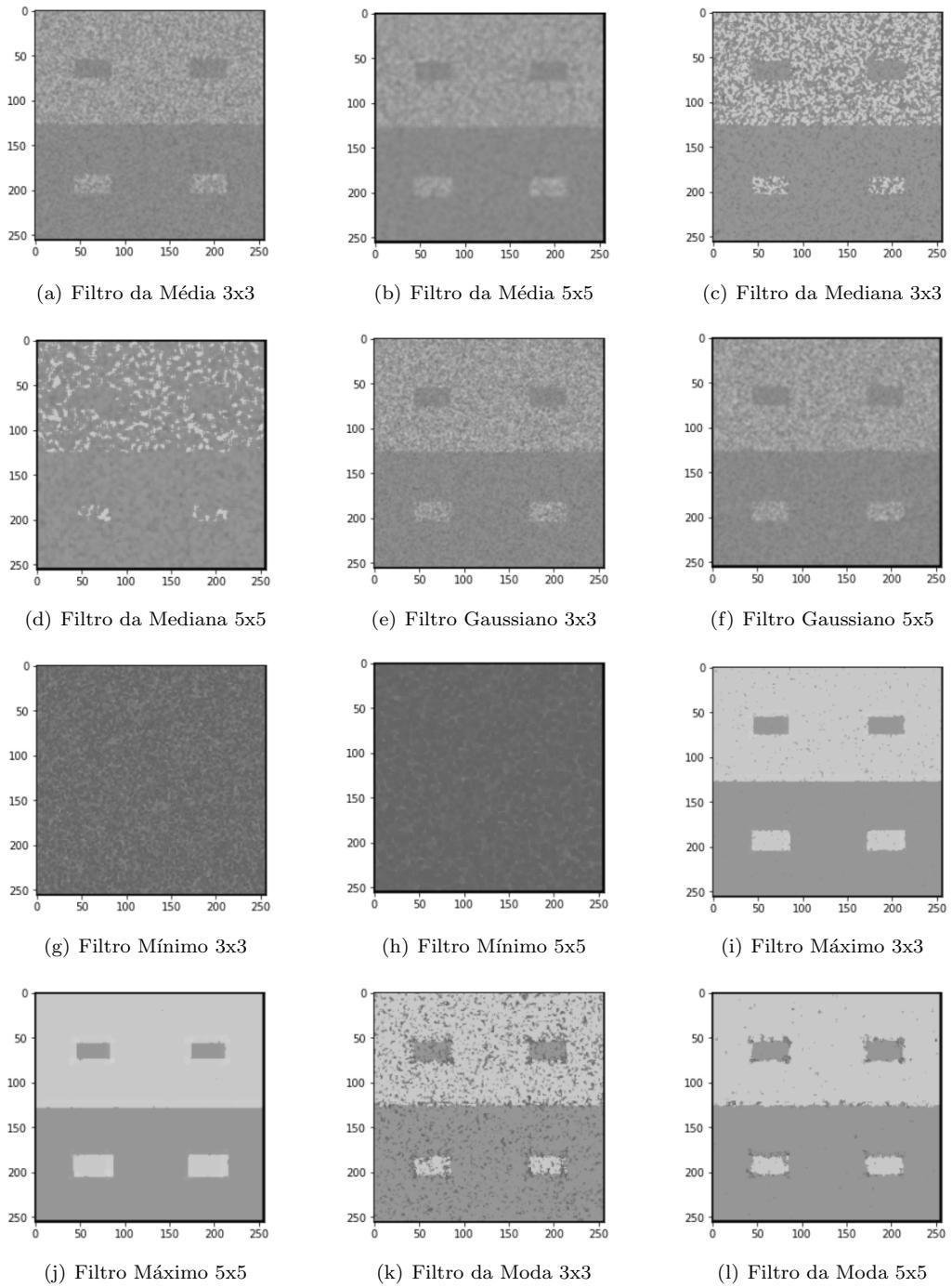


Figura 20: Imagem (c) com ruído uniforme suavizada pelos filtros.

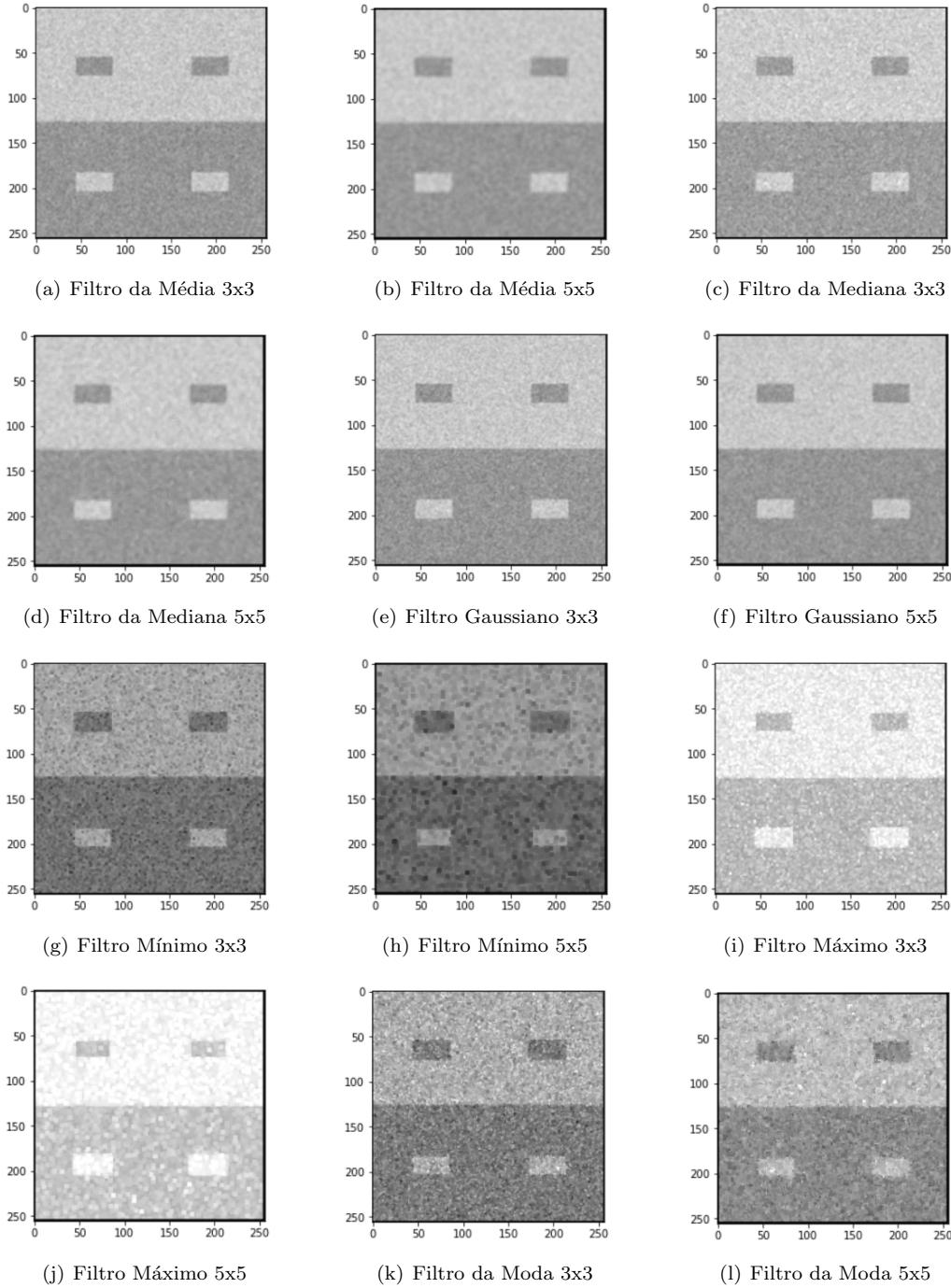


Figura 21: Imagem (c) com ruído gaussiano suavizada pelos filtros.

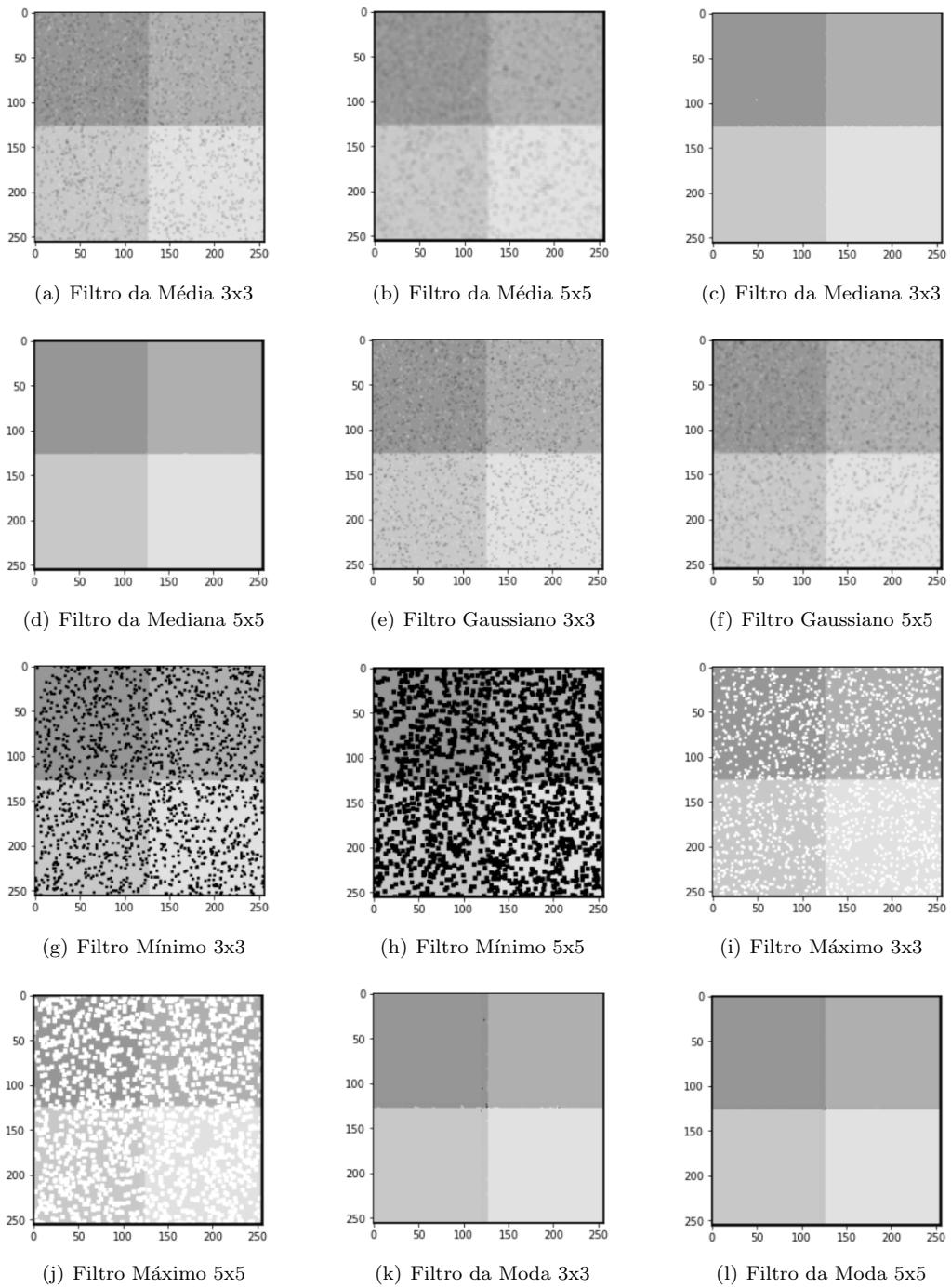


Figura 22: Imagem (d) com ruído sal e pimenta suavizada pelos filtros.

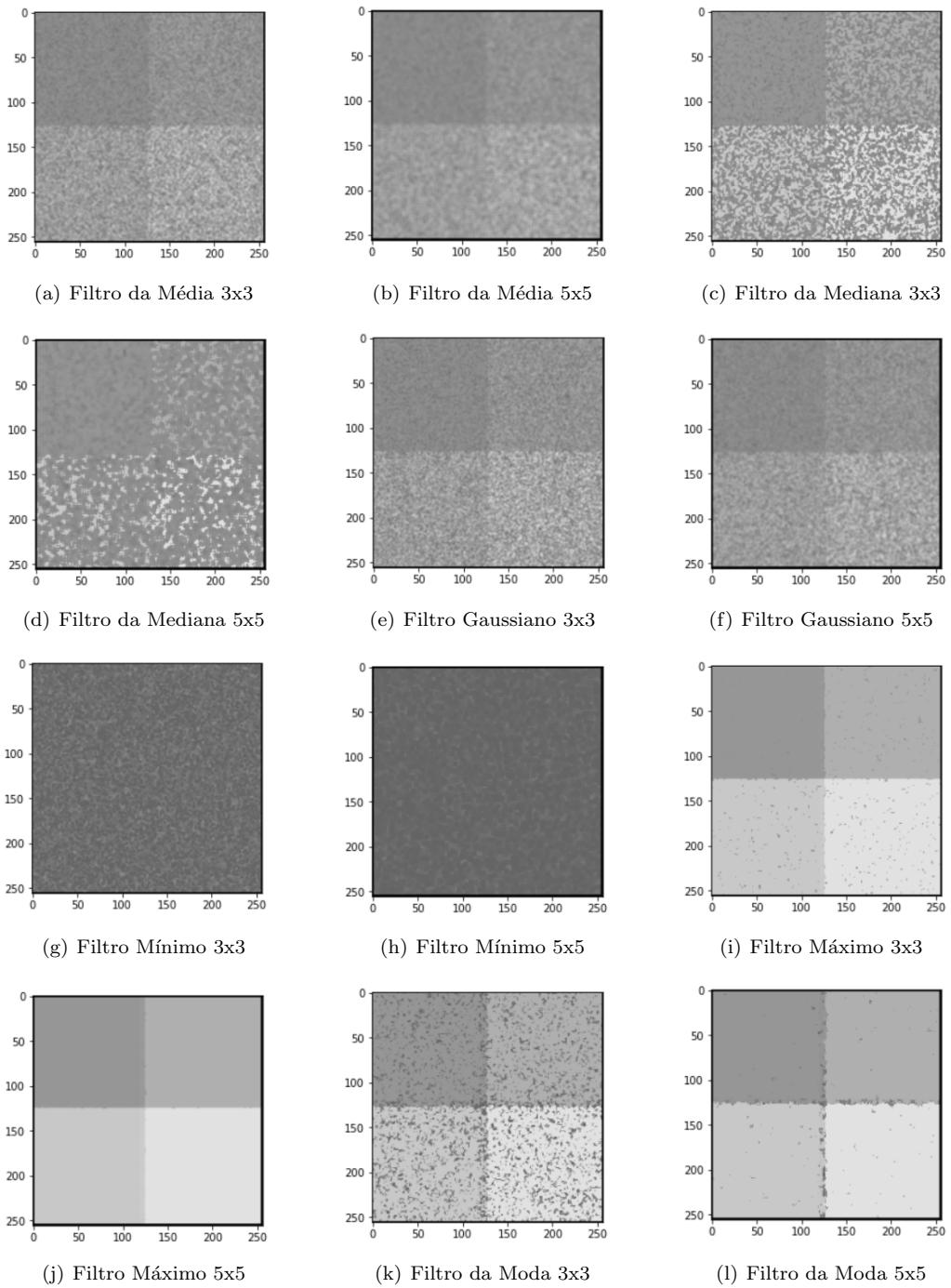


Figura 23: Imagem (d) com ruído uniforme suavizada pelos filtros.

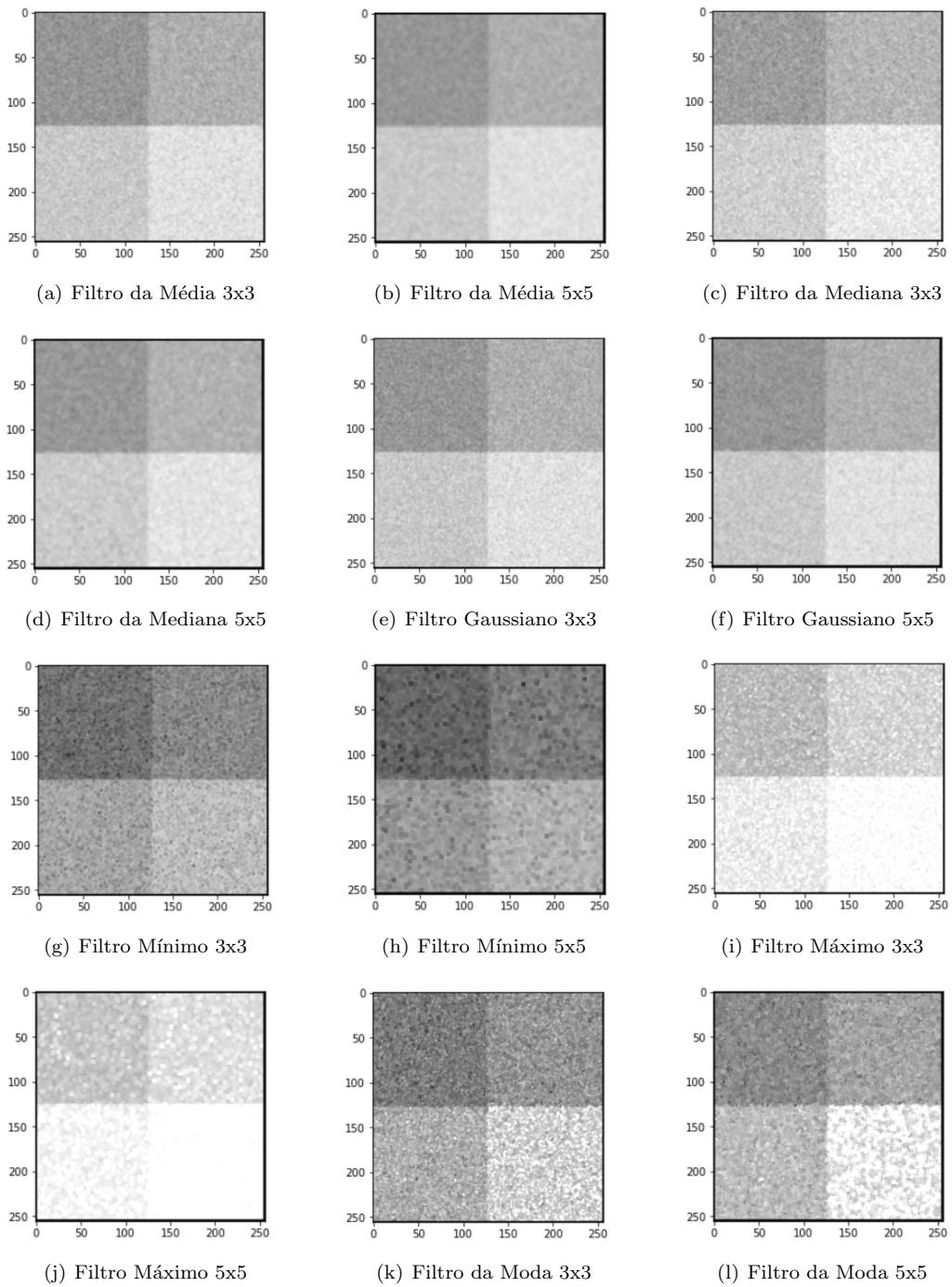


Figura 24: Imagem (d) com ruído gaussiano suavizada pelos filtros.

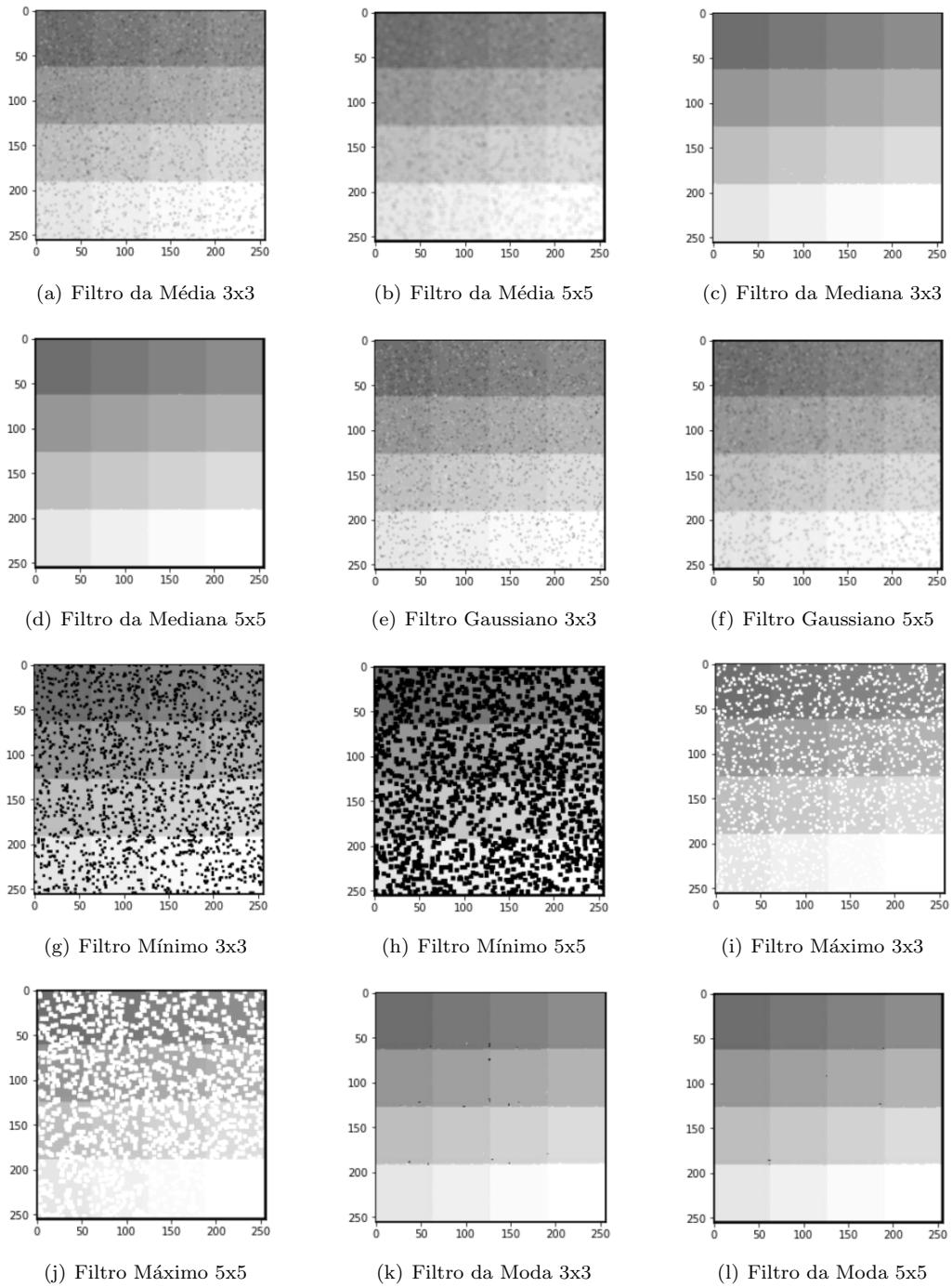


Figura 25: Imagem (e) com ruído sal e pimenta suavizada pelos filtros.

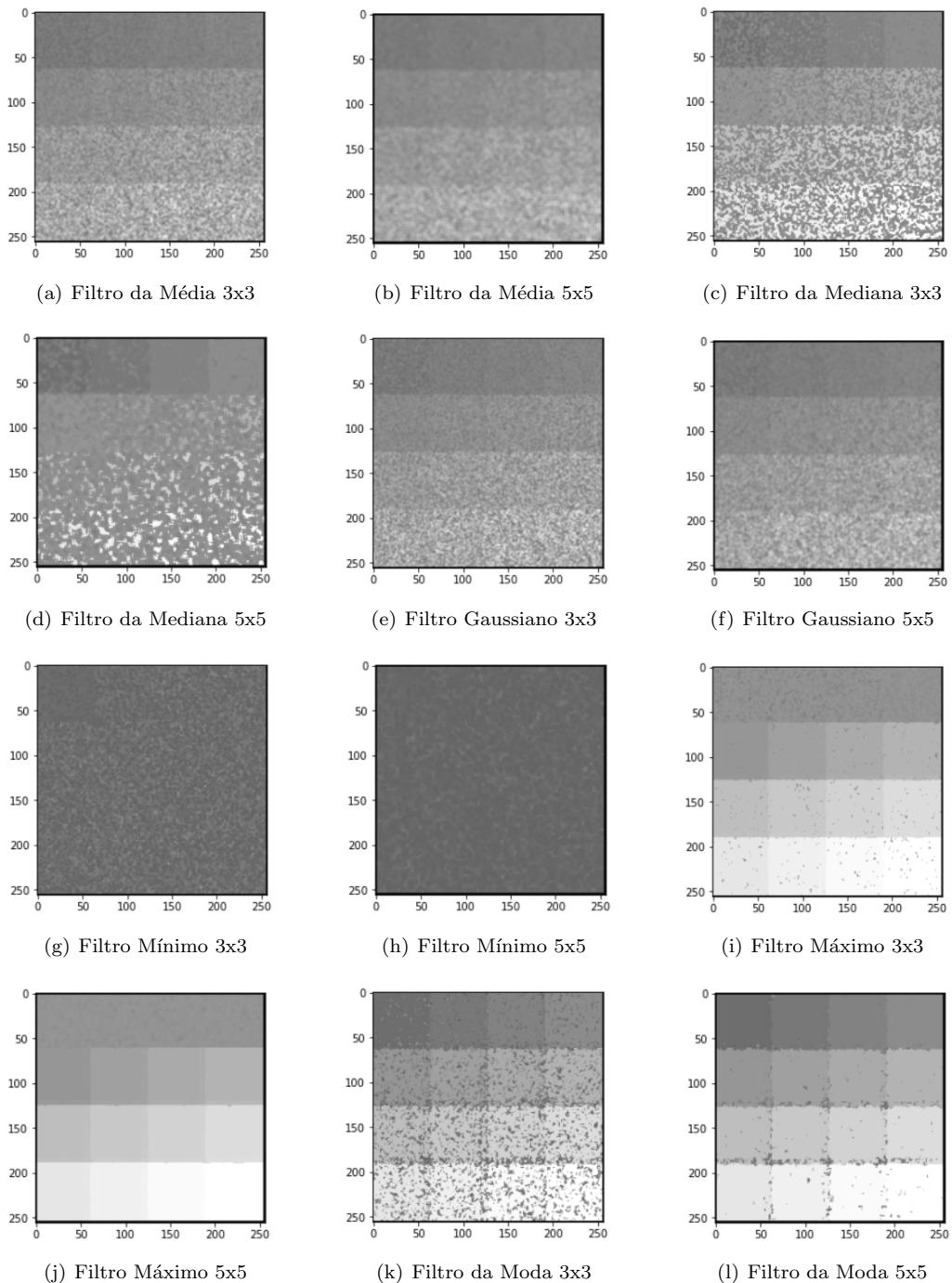


Figura 26: Imagem (e) com ruído uniforme suavizada pelos filtros.

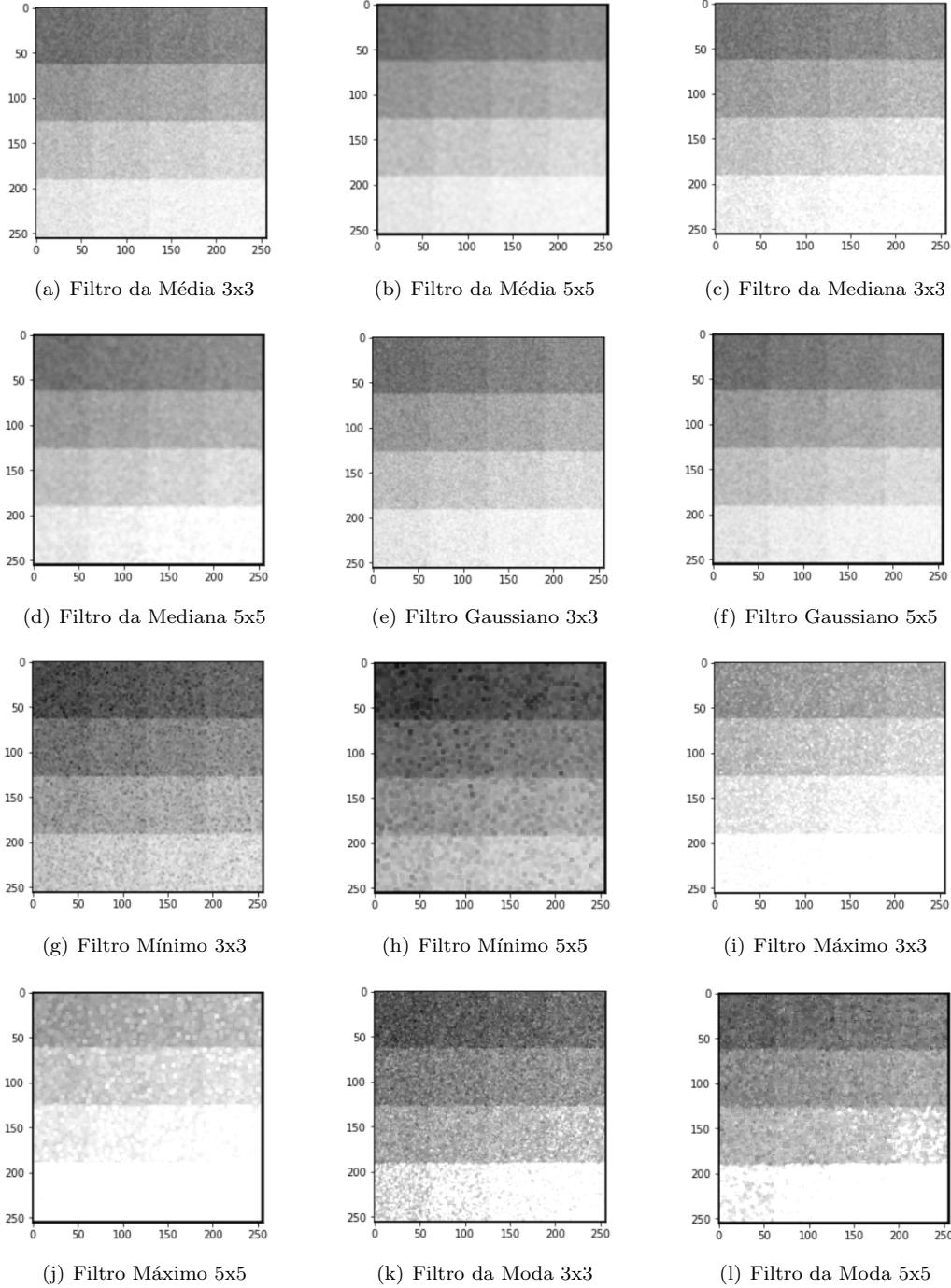


Figura 27: Imagem (e) com ruído gaussiano suavizada pelos filtros.

**b)** Utilizando as máscaras  $h_1$  e  $h_2$  para aplicar o filtro passa-alta normalizado nas imagens com ruídos, obtivemos os resultados abaixo.

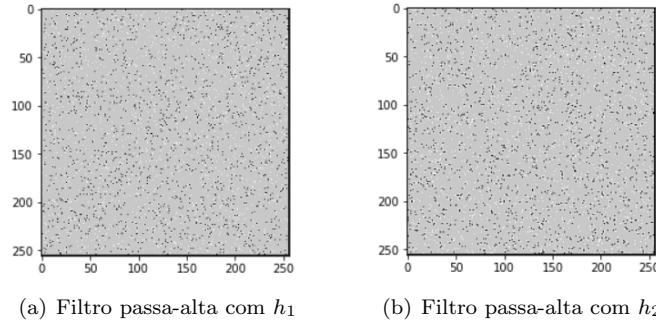


Figura 28: Imagem (a) com ruído sal e pimenta suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

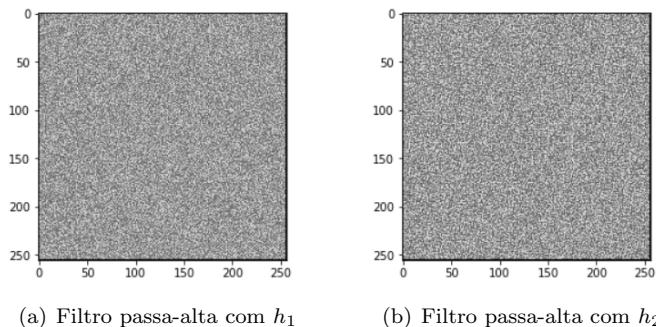


Figura 29: Imagem (a) com ruído uniforme suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

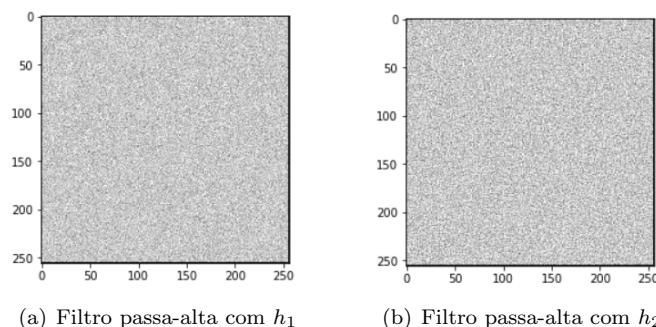


Figura 30: Imagem (a) com ruído gaussiano suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

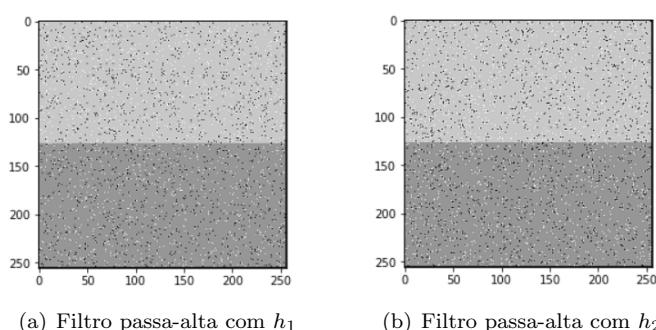


Figura 31: Imagem (b) com ruído sal e pimenta suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

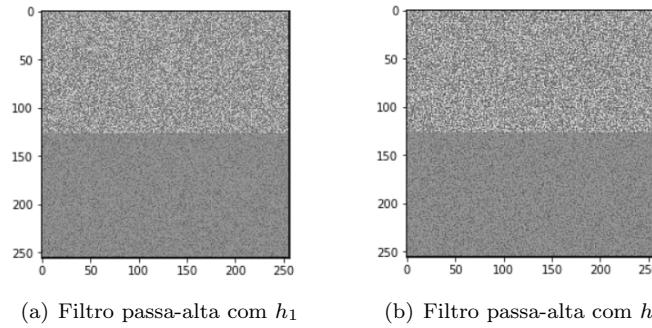


Figura 32: Imagem (b) com ruído uniforme suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

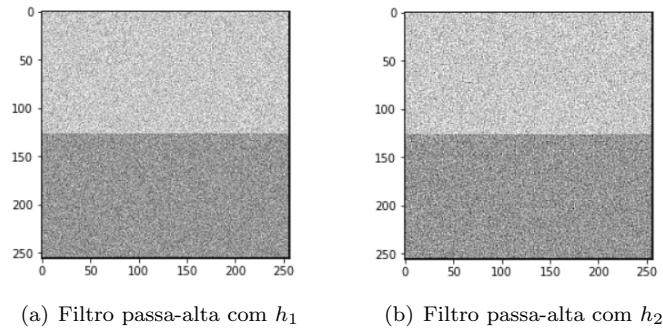


Figura 33: Imagem (b) com ruído gaussiano suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

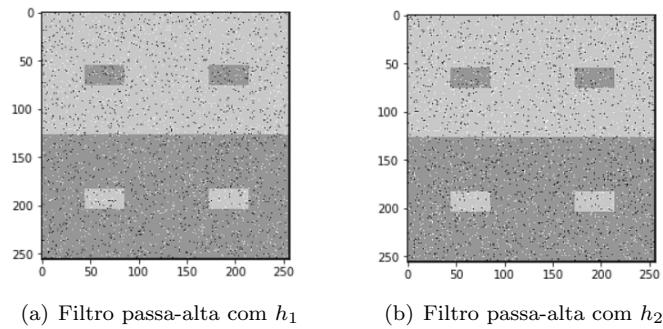


Figura 34: Imagem (c) com ruído sal e pimenta suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

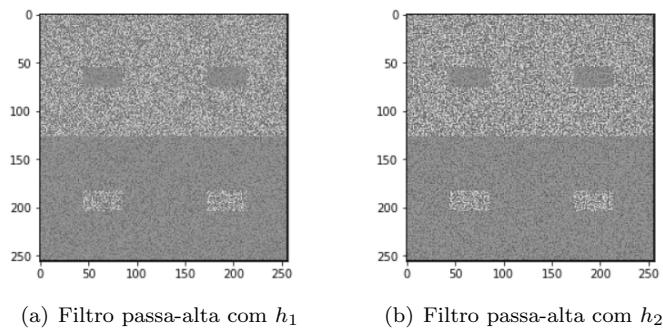


Figura 35: Imagem (c) com ruído uniforme suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

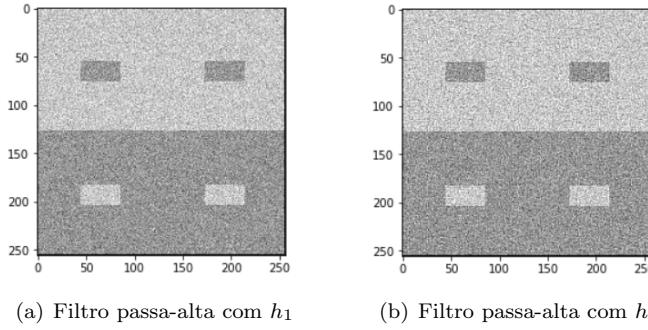


Figura 36: Imagem (c) com ruído gaussiano suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

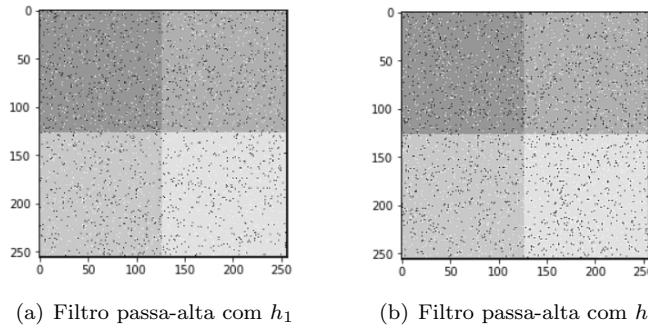


Figura 37: Imagem (d) com ruído sal e pimenta suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

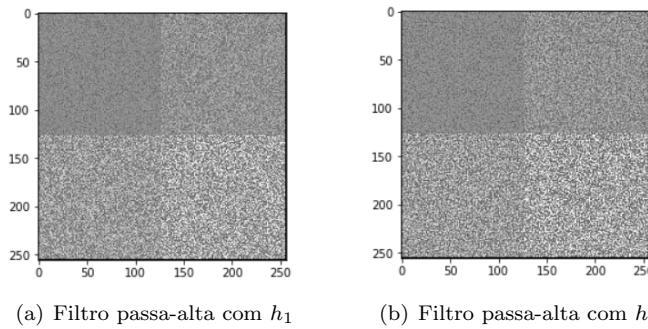


Figura 38: Imagem (d) com ruído uniforme suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

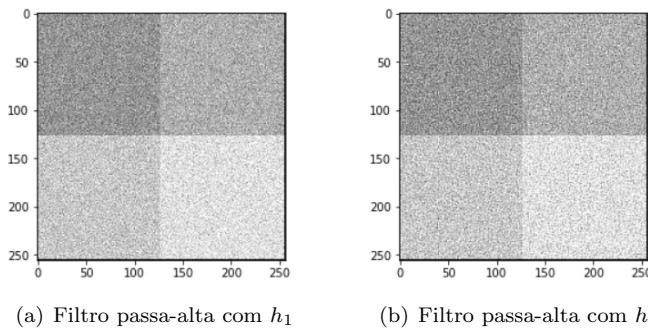


Figura 39: Imagem (d) com ruído gaussiano suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

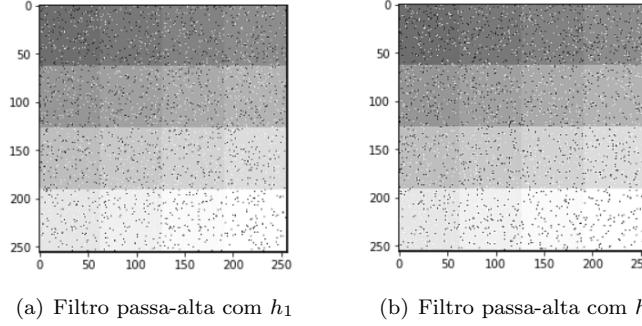


Figura 40: Imagem (e) com ruído sal e pimenta suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

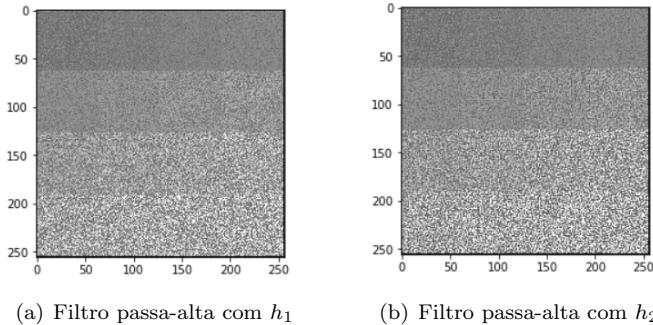


Figura 41: Imagem (e) com ruído uniforme suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

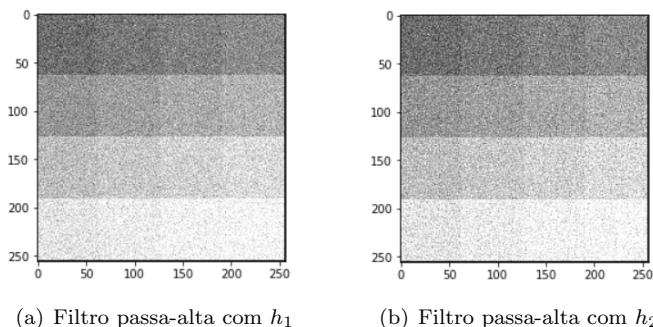


Figura 42: Imagem (e) com ruído gaussiano suavizada pelo filtro passa-alta com as máscaras  $h_1$  e  $h_2$ .

c) Para cada imagem ruídos, foram aplicados os filtros e visualmente foram avaliados os melhores resultados. A partir disso, foram retiradas as métricas de erro para a avaliação. As métricas escolhidas foram o Erro médio absoluto, Erro médio quadrático e Raiz do erro médio quadrático. O código abaixo mostra a extração dos erros:

```
# Calcula o erro médio absoluto entre as duas imagens
sklearn.metrics.mean_absolute_error(img, filter_img)

# Calcula o erro médio quadrático entre as duas imagens
sklearn.metrics.mean_squared_error(img, filter_img)

# Calcula a raíz do erro médio quadrático entre as duas imagens
root_mean_square_error(img, filter_img)
```

Para a imagem (a), após aplicar os ruídos, os melhores resultados de filtragem se encontram na tabela 2.

Tabela 2: Melhores resultados para a imagem (a)

Ruído + Filtro	Métrica	Erro
Sal e pimenta + Filtro da mediana 3x3	Erro médio absoluto	3.11
	Erro médio quadrático	622.60
	Raíz do erro médio quadrático	24.95
Uniforme + Filtro máximo 5x5	Erro médio absoluto	11.47
	Erro médio quadrático	622.60
	Raíz do erro médio quadrático	35.85
Gaussiano + Filtro gaussiano 5x5	Erro médio absoluto	11.47
	Erro médio quadrático	1284.98
	Raíz do erro médio quadrático	35.21

Para a imagem (b), após aplicar os ruídos, os melhores resultados de filtragem se encontram na tabela 3.

Tabela 3: Melhores resultados para a imagem (b)

Ruído + Filtro	Métrica	Erro
Sal e pimenta + Filtro da mediana 5x5	Erro médio absoluto	5.47
	Erro médio quadrático	968.37
	Raíz do erro médio quadrático	31.11
Uniforme + Filtro máximo 5x5	Erro médio absoluto	5.85
	Erro médio quadrático	985.79
	Raíz do erro médio quadrático	31.40
Gaussiano + Filtro gaussiano 5x5	Erro médio absoluto	10.81
	Erro médio quadrático	1015.00
	Raíz do erro médio quadrático	31.86

Para a imagem (c), após aplicar os ruídos, os melhores resultados de filtragem se encontram na tabela 4.

Tabela 4: Melhores resultados para a imagem (c)

Ruído + Filtro	Métrica	Erro
Sal e pimenta + Filtro da mediana 5x5	Erro médio absoluto	5.61
	Erro médio quadrático	970.29
	Raíz do erro médio quadrático	31.15
Uniforme + Filtro máximo 5x5	Erro médio absoluto	6.75
	Erro médio quadrático	1023.84
	Raíz do erro médio quadrático	31.99
Gaussiano + Filtro gaussiano 5x5	Erro médio absoluto	11.01
	Erro médio quadrático	1019.98
	Raíz do erro médio quadrático	31.94

Para a imagem (d), após aplicar os ruídos, os melhores resultados de filtragem se encontram na tabela 5.

Tabela 5: Melhores resultados para a imagem (d)

Ruído + Filtro	Métrica	Erro
Sal e pimenta + Filtro da mediana 5x5	Erro médio absoluto	5.92
	Erro médio quadrático	1116.63
	Raíz do erro médio quadrático	33.41
Uniforme + Filtro máximo 5x5	Erro médio absoluto	6.49
	Erro médio quadrático	1139.47
	Raíz do erro médio quadrático	33.76
Gaussiano + Filtro gaussiano 5x5	Erro médio absoluto	11.26
	Erro médio quadrático	1164.45
	Raíz do erro médio quadrático	34.12

Para a imagem (e), após aplicar os ruídos, os melhores resultados de filtragem se encontram na tabela 6.

Tabela 6: Melhores resultados para a imagem (e)

Ruído + Filtro	Métrica	Erro
Sal e pimenta + Filtro da mediana 5x5	Erro médio absoluto	5.96
	Erro médio quadrático	1147.69
	Raíz do erro médio quadrático	33.88
Uniforme + Filtro máximo 5x5	Erro médio absoluto	8.63
	Erro médio quadrático	759.43
	Raíz do erro médio quadrático	27.56
Gaussiano + Filtro gaussiano 5x5	Erro médio absoluto	11.63
	Erro médio quadrático	1202.18
	Raíz do erro médio quadrático	34.67

## 4 Exercícios aula 7 - Modelo de Cores e Fatiamento

### 4.1 Exercício 3)

3) Escreva um programa que receba as imagens abaixo, converta para o padrão HSI e aplique o processo de transformações de cores: equalização de histograma. Apresentar os histogramas equalizados. Observe os resultados dos exercícios 2 e 3. Indique, se possível, qual estratégia forneceu os melhores resultados e justifique sua resposta.



Figura 43: Imagens do enunciado.

Para o desenvolvimento do exercício foi definida uma função para exibir a imagem, seu histograma e histograma cumulativo. Para essa função, destacam-se as seguintes linhas de código:

```
...
ax_img.imshow(image, cmap=plt.cm.gray) #exibindo a imagem
...
ax_hist.hist(image.ravel(), bins=bins, color='black') #exibindo o histograma
...
img_cdf, bins = exposure.cumulative_distribution(image, bins) #calculando o histograma cumulativo
ax_cdf.plot(bins, img_cdf, 'r') #exibindo
```

Em seguida, outra função foi definida para aplicar a função anterior na imagem original, imagem equalizada e na imagem equalizada no canal V do modelo HSV. Nessa função, as partes principais são:

```
...
#capturando os plots para a imagem original
ax_img, ax_hist, ax_cdf = plot_img_and_hist(img, axes[:, 0])
```

```
...
#equalizando a imagem
img_eq = exposure.equalize_hist(img)
#capturando os plots para a imagem equalizada
ax_img, ax_hist, ax_cdf = plot_img_and_hist(img_eq, axes[:, 1])
...
#convertendo a imagem para hsv
hsv_img = rgb2hsv(img)
#equalizando o canal V, apenas
hsv_img[:, :, 2] = exposure.equalize_hist(hsv_img[:, :, 2])
#capturando os plots para a imagem em rgb (RGB -> HSV -> equaliza V -> RGB)
ax_img, ax_hist, ax_cdf = plot_img_and_hist(hsv2rgb(hsv_img), axes[:, 2])
...
```

As figuras a seguir apresentam o resultado da aplicação dessa função para as três imagens solicitadas.

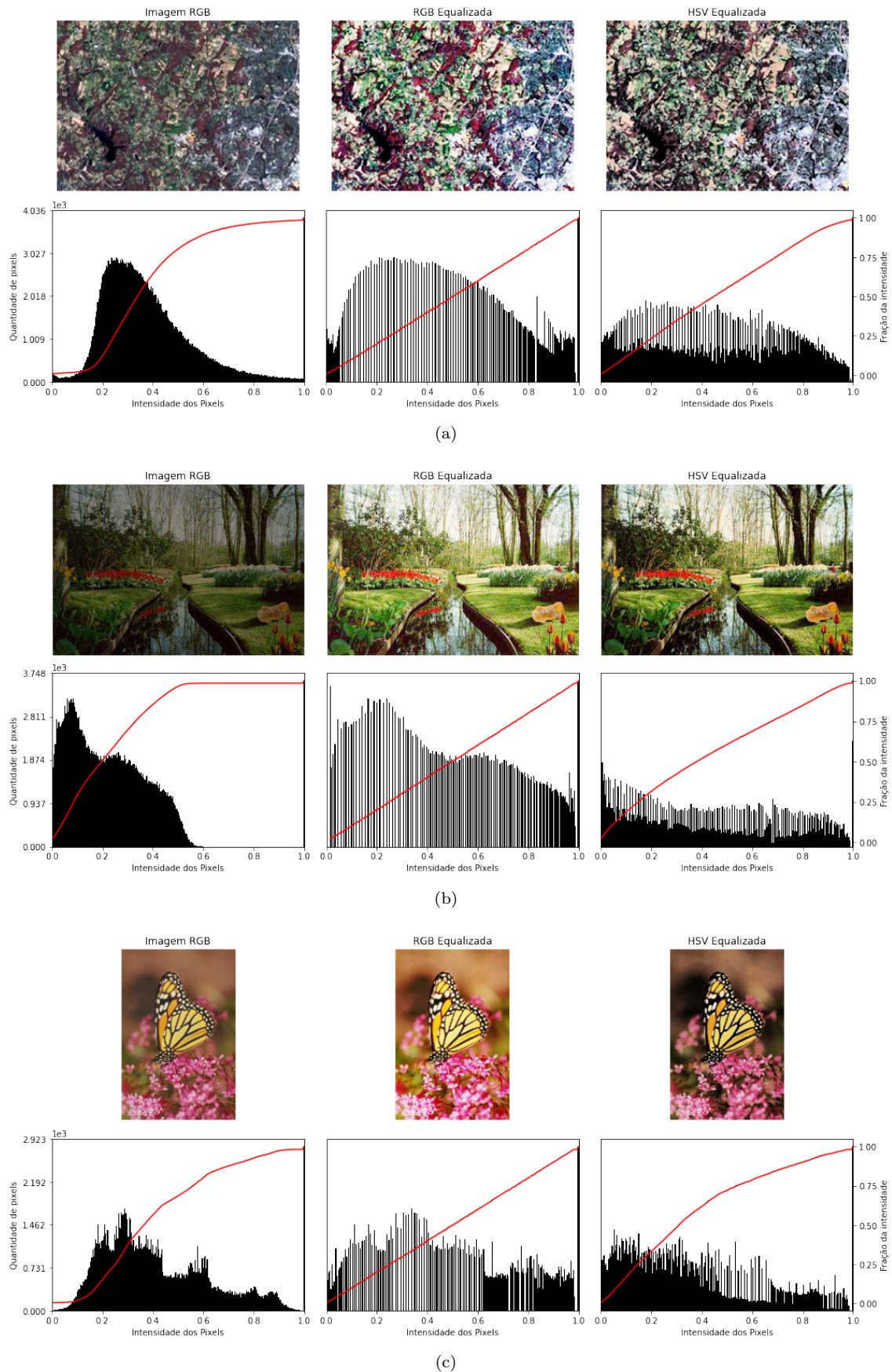


Figura 44: Resultados obtidos.

Ao realizar a conversão para HSV, equalizando o canal V e voltando para RGB, foi possível analisar que o resultado é visualmente melhor. O histograma cumulativo não é uma função identidade, mas analisando a distribuição das intensidades houve um melhor resultado. Nesse caso, como foi utilizado o modelo HSV, a informação de brilho/luminância foi equalizada e não houve distorção nas informações de cromaticidade.

## 4.2 Exercício 5)

- 5) Considere a imagem indicada abaixo (Padronizada em RGB). Crie um programa para realizar o processo de fatiamento por intensidades. Cada intensidade deve ser associada a uma cor.



Figura 45: Imagem do enunciado.

Para o desenvolvimento desse exercício foram utilizadas duas abordagens: realizar o processo por meio da substituição de cada intensidade por uma matiz, no caso V do canal HSV; realizar o processo por meio da atribuição de cores aos pixels. Na primeira abordagem, destacam-se as seguintes linhas de código:

```
...
hsv_img = rgb2HSV(img5) #convertendo a imagem para hsv
hue_img = hsv_img[:, :, 2] #capturando o V
...
ax0.imshow(img5)
...
ax1.imshow(hue_img, cmap='Spectral_r')
...
```

Analizando visualmente o resultado, temos:

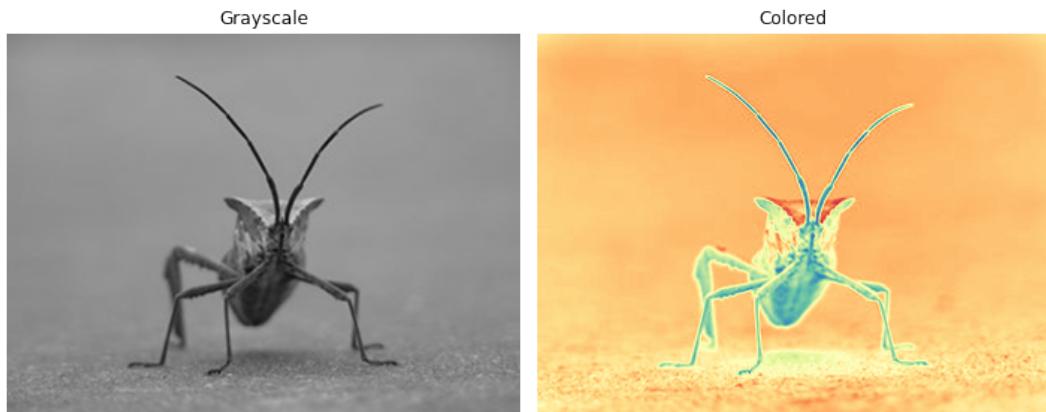


Figura 46: Resultado obtido pela primeira abordagem

Já na segunda abordagem foi criada uma função que itera pelos pixens da imagem, analisa a intensidade com o valor do limiar de corte e atribui uma cor de acordo com esse valor. A função a apresentada a seguir.

```
def our_digitize(image, thresholds, colors):
    row, column = image.shape
    #criando uma imagem vazia para armazenar o resultado
    regions = np.zeros((row, column, 3))

    #iterando sobre as linhas i e colunas j da imagem
    for i in range(row):
        for j in range(column):
            #iterando sobre os pontos de cortes e verificando se a
            #intensidade do pixel para atribuir a classe ao resultado
            for k in range(len(thresholds) - 1):
                if thresholds[k] <= image[i, j] < thresholds[k + 1]:
                    regions[i, j, :] = colors[k]

    return regions
```

Essa função foi aplicada na imagem solicitada da seguinte maneira:

```
img5 = imread('/content/Img5.bmp', as_gray=True)*255
img5_fat = our_digitize(image = img5,
                        thresholds = [0, 65, 80, 120, 150, 170, 210, 256],
                        colors = [[255,0,0], [255,255,0], [0,255,0], [0,255,255],
                                  [0,125,255], [0,0,255], [0,0,125]])
```

Dessa forma, para pixens dentro dos intervalos definidos pelos valores de intensidade passados como parâmetro, foram atribuídas as respectivas cores (nesse caso as cores estavam na organização BGR). O resultado obtido é apresentado na figura a seguir.

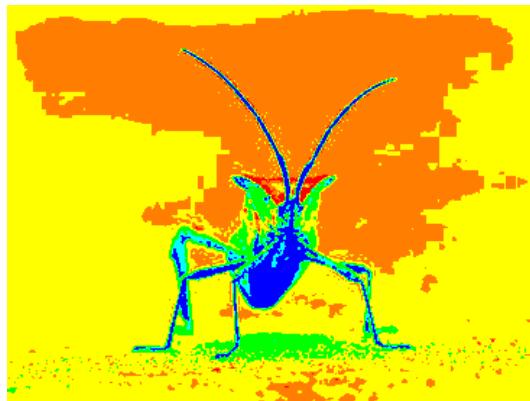


Figura 47: Resultado obtido pela segunda abordagem

## 5 Exercícios aula 8 - Filtragem no domínio da frequência

### 5.1 Exercício 4)

- 4) Aplique os ruídos impulsivo e gaussianos sobre duas imagens monocromáticas (exemplificadas no exercício 2). Em seguida, implemente e execute a DFT sobre as imagens com ruídos. Plete o espectro de Fourier. Aplique dois filtros no domínio da frequência para suavizar os ruídos inseridos previamente. Apresente os espectros de Fourier antes e após as inserções de ruídos, bem como as imagens reconstruídas após os processos de filtragem. Explique detalhadamente cada etapa e os filtros propostos.

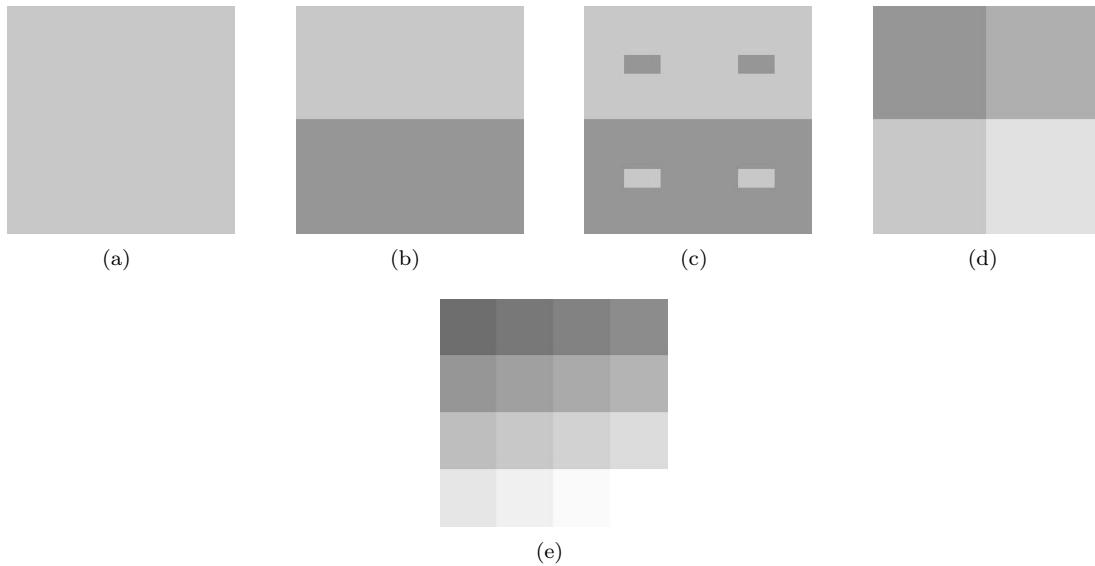


Figura 48: Imagens do exercício 2.

Imagens escolhidas:

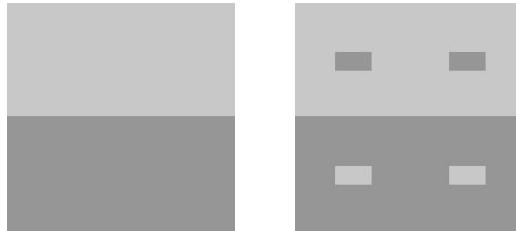


Figura 49: Imagens escolhidas.

**Resposta:** Para aplicar os ruídos impulsivo e gaussiano, foram utilizados os seguintes códigos:

```
# Para o ruído impulsivo (sal e pimenta)
# Imagem (b)
image_b_noised_sp = skimage.util.random_noise(image_b, mode='s&p', amount=0.05)*255
plt.imshow(image_b_noised_sp, cmap='gray', vmin=0, vmax=255)
# Imagem (c)
image_c_noised_sp = skimage.util.random_noise(image_c, mode='s&p', amount=0.05)*255
plt.imshow(image_c_noised_sp, cmap='gray', vmin=0, vmax=255)

# Para o ruído gaussiano
# Imagem (b)
image_b_noised_gaussian = skimage.util.random_noise(image_b, mode='gaussian', mean=0, var=0.1)*255
plt.imshow(image_b_noised_gaussian, cmap='gray', vmin=0, vmax=255)
# Imagem (c)
image_c_noised_gaussian = skimage.util.random_noise(image_c, mode='gaussian', mean=0, var=0.1)*255
plt.imshow(image_c_noised_gaussian, cmap='gray', vmin=0, vmax=255)
```

A aplicação do ruído gerou as seguintes imagens a baixo:

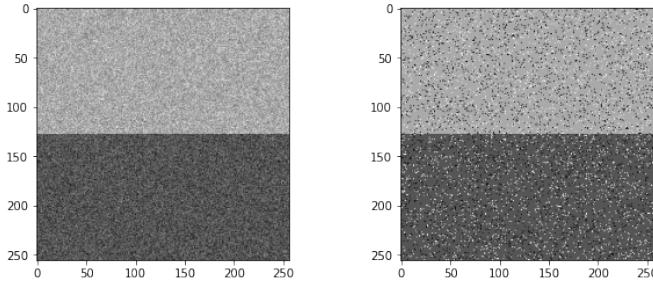


Figura 50: Imagem (b) escolhida para o exercício 4 após a aplicação dos ruídos.

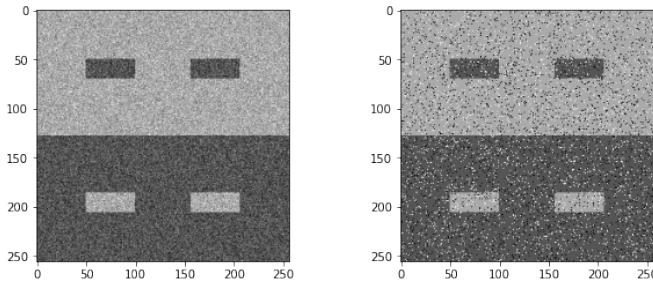


Figura 51: Imagem (c) escolhida para o exercício 4 após a aplicação dos ruídos.

O próximo passo é aplicar a DFT sobre as imagens com ruído apresentadas acima. Para isso, foram aplicados os códigos abaixo:

Para a imagem (b):

```
# Com ruído impulsivo
image_b_noised_sp_dft = np.fft.fftshift(np.fft.fft2(image_b_noised_sp))
plt.imshow(np.log(abs(image_b_noised_sp_dft)), cmap='gray')

# Com ruído gaussiano
image_b_noised_gaussian_dft = np.fft.fftshift(np.fft.fft2(image_b_noised_gaussian))
plt.imshow(np.log(abs(image_b_noised_gaussian_dft)), cmap='gray')
```

Resultado:

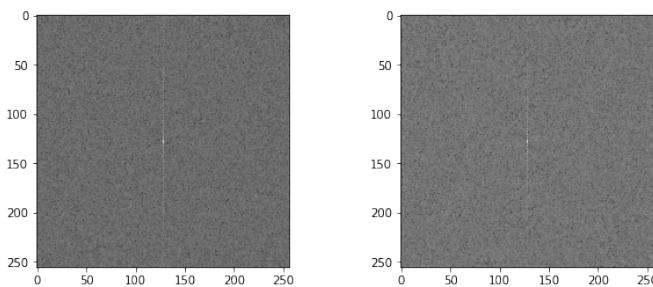


Figura 52: Imagem (b) com ruído sal e pimenta e gaussiano após a aplicação da DFT.

Para a imagem (c):

```
# Com ruído impulsivo
```

```

image_c_noised_sp_dft = np.fft.fftshift(np.fft.fft2(image_c_noised_sp))
plt.imshow(np.log(abs(image_c_noised_sp_dft)), cmap='gray')

# Com ruído gaussiano
image_c_noised_gaussian_dft = np.fft.fftshift(np.fft.fft2(image_c_noised_gaussian))
plt.imshow(np.log(abs(image_c_noised_gaussian_dft)), cmap='gray')

```

Resultado:

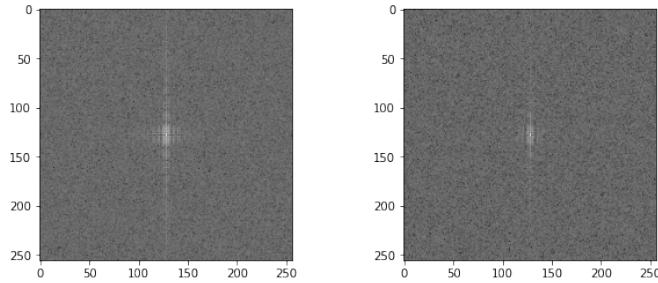


Figura 53: Imagem (c) com ruído sal e pimenta e gaussiano após a aplicação da DFT.

Agora, vamos verificar o Espectro de Fourier das imagens originais:

```

image_b_dft = np.fft.fftshift(np.fft.fft2(image_b))
plt.imshow(np.log(abs(image_b_dft)), cmap='gray')

image_c_dft = np.fft.fftshift(np.fft.fft2(image_c))
plt.imshow(np.log(abs(image_c_dft)), cmap='gray')

```

Resultado:

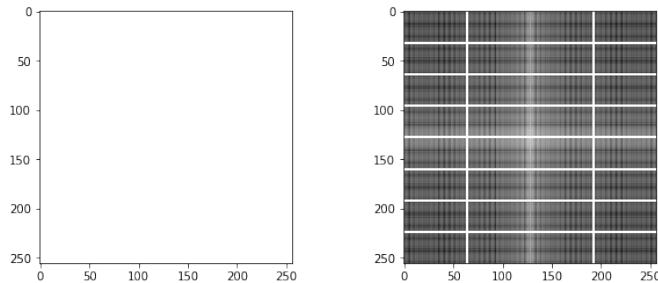


Figura 54: Espectro de Fourier das imagens originais.

Agora, vamos verificar o Espectro de Fourier das imagens com ruído:

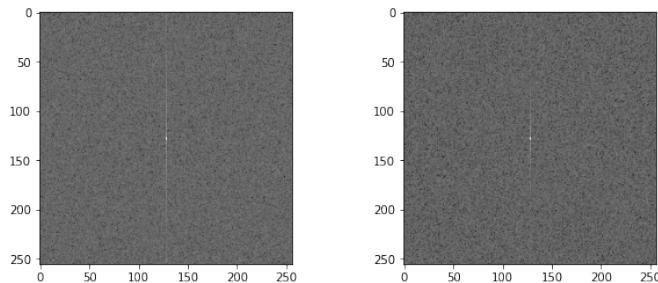


Figura 55: Espectro de Fourier da imagem (b) com os ruídos impulsivo e gausiano.

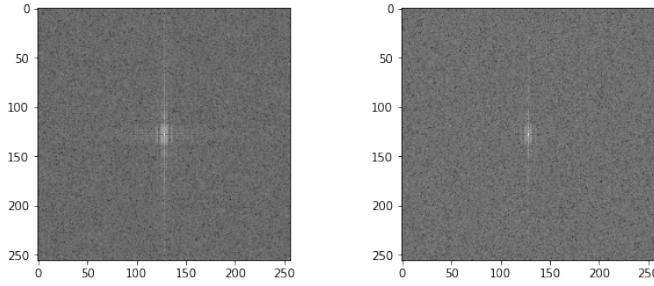


Figura 56: Espectro de Fourier da imagem (c) com os ruídos impulsivo e gausiano.

Por fim, aplicaremos os filtros no domínio da frequência:

Passa-Baixa Butterworth código:

```
# Butterworth Passa-Baixa (criação do filtro)
linhas, colunas = image_b.shape
mascara = np.zeros((linhas, colunas))
raio = 3
centro = [int(linhas/2), int(colunas/2)]
x, y = np.ogrid[:linhas, :colunas]
circulo = (x - centro[0]) ** 2 + (y - centro[1]) ** 2 <= raio*raio
mascara[circulo] = 1 # filtro passa baixa
plt.imshow(mascara, cmap='gray')
```

Passa-Baixa Butterworth em (b):

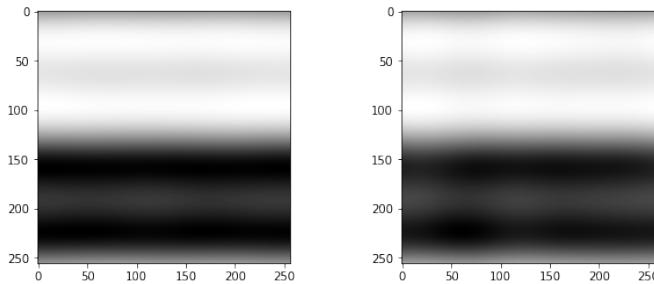


Figura 57: Filtro passa-baixa Butterworth aplicado na imagem (b) com ruído impulsivo e gaussiano.

Passa-Baixa Butterworth em (c):

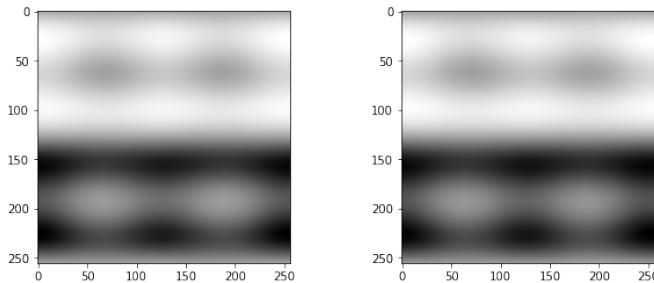


Figura 58: Filtro aplicado na imagem (c) com ruído impulsivo e gaussiano.

Filtro gaussiano no domínio da frequência:

```

# Dominio da frequencia
FDF = np.fft.fftshift(img_freq_dom)

# Filtro Gaussiano domínio da frequência
I,J = img_freq_dom.shape
filter = np.zeros((I,J), dtype=np.float32)
D0 = 15
for i in range(I):
    for j in range(J):
        # Passa-Baixa
        D = np.sqrt((i-I/2)**2 + (j-J/2)**2)
        filter[i,j] = np.exp(-D**2/(2*D0*D0))

# Imagem no domínio da frequência * filtro gaussiano
result = FDF * filter
TF1 = np.fft.ifftshift(result)
g = np.abs(np.fft.ifft2(TF1))

```

Mostrando a imagem na tela:

```

# Plotando a imagem
plt.imshow(g, cmap='gray')
plt.show()

```

Resultados:

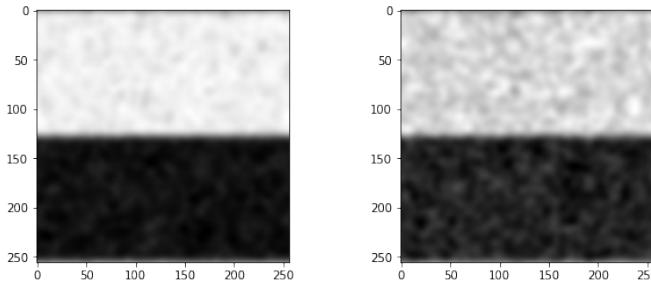


Figura 59: Gaussiano passa-baixa aplicado na imagem (b) com ruído impulsivo e gaussiano.

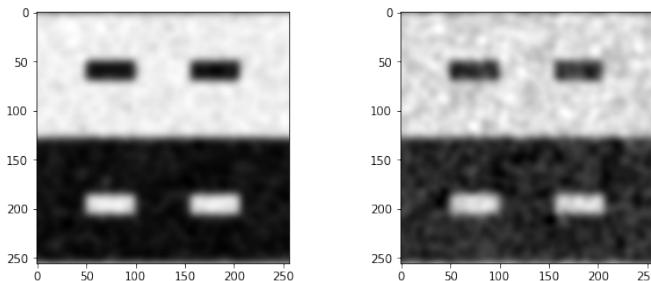


Figura 60: Gaussiano passa-baixa aplicado na imagem (c) com ruído impulsivo e gaussiano.

## 6 Exercícios aula 9 - Segmentação de imagens

### 6.1 Exercício 2)

2) Implemente e execute a técnica de Otsu sobre a imagem indicada à baixo. Inicialmente, o algoritmo deve ser capaz de separar duas classes (fundo e objeto). Em seguida, modifique o código para obter

três classes como resultado. Apresente os valores dos limiares e as imagens que definem cada uma das classes. As máscaras devem ser aplicadas para segmentar regiões coloridas que definem cada classe.

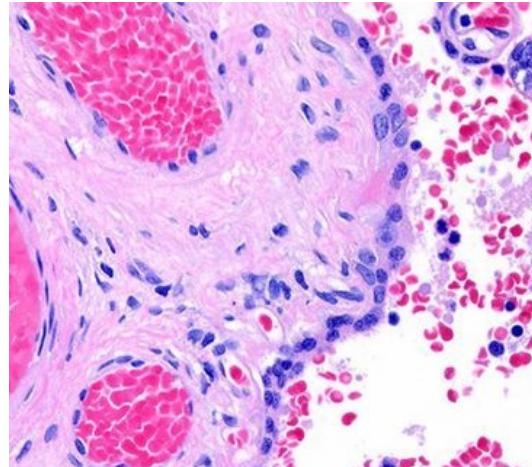


Figura 61: Imagem do enunciado.

A função abaixo implementa o método de Otsu para duas classes. Nela é feita uma busca pelo limiar que minimiza a variância entre as classes definidas pelo limiar de corte.

```
def our_threshold_multiotstu_2(image):
    #serão testados todos os limiares de 0 a maior intensidade da imagem
    threshold_range = range(np.max(image) + 1)
    criterias = []

    for th in threshold_range: #para cada intensidade
        thresholded_im = np.zeros(image.shape) #cria uma imagem vazia
        thresholded_im[image >= th] = 1 #e segmenta a imagem de acordo com o limiar

        nb_pixels = image.size #quantidade de pixels
        nb_pixels1 = np.count_nonzero(thresholded_im) #quantidade de pixels maiores ou iguais ao limiar
        weight1 = nb_pixels1 / nb_pixels #probabilidade do pixel cair em uma classe
        weight0 = 1 - weight1 #probabilidade do pixel cair na outra classe

        if weight1 == 0 or weight0 == 0: #caso uma das classes estejam vazias
            criterias.append(10000)

        val_pixels1 = image[thresholded_im == 1] #encontrando os pixels que pertence a cada classe
        val_pixels0 = image[thresholded_im == 0]

        var0 = np.var(val_pixels0) if len(val_pixels0) > 0 else 0 #calculando a variancia das classes
        var1 = np.var(val_pixels1) if len(val_pixels1) > 0 else 0

        #variancia entre classes: soma ponderada entre peso e variancia
        criterias.append(weight0 * var0 + weight1 * var1)

    #o melhor limiar é aquele que minimiza a variancia entre-classes
    best_threshold = threshold_range[np.argmin(criterias)]

    return best_threshold, criterias
```

Essa outra função foi criada para o método de Otsu para três classes. Seu conceito de funcionamento é semelhante ao anterior. Nesse caso existem dois limiares para serem testados.

```

def our_threshold_multotsu_3(image): #abordagem um pouco diferente da funcao anterior
    max_value = np.max(image)
    hist, bins = np.histogram(image, bins=max_value) #calcula o histograma da imagem
    p = hist / image.size #histograma normalizado

    thresholds = []
    variances = []

    th_1 = 0

    while th_1 <= (max_value - 1): #selecionando o primeiro limiar
        supposed_th = []
        supposed_variances = []

        th_2 = th_1 + 1

        while th_2 <= max_value: #selecionando o segundo limiar (sempre a frente do primeiro)
            p_1 = np.sum(p[:th_1]) #calculando a probabilidade do pixel cair em cada classe
            p_2 = np.sum(p[th_1:th_2])
            p_3 = np.sum(p[th_2:])

            if p_1 == 0 or p_2 == 0 or p_3 == 0 :
                th_2 += 1
                continue

            #calculando a variancia de cada classe
            m_1 = (1/p_1) * np.sum([i*p_i for i, p_i in zip(p[:th_1], np.arange(0, th_1))])
            m_2 = (1/p_2) * np.sum([i*p_i for i, p_i in zip(p[th_1:th_2], np.arange(th_1, th_2))])
            m_3 = (1/p_3) * np.sum([i*p_i for i, p_i in zip(p[th_2:], np.arange(th_2, max_value))])

            m_g = p_1*m_1 + p_2*m_2 + p_3*m_3 #intensidade global média

            #calculando a variancia entre as classes
            class_variance = p_1*(m_1 - m_g)**2 + p_2*(m_2 - m_g)**2 + p_3*(m_3 - m_g)**2

            supposed_variances.append(class_variance)
            supposed_th.append((th_1, th_2))

            th_2 += 1

        #depois de testar supostos par de limires para cada um do primeiro limiar,
        #obtem o melhor de cada teste
        if len(supposed_variances) > 0:
            variances.append(max(supposed_variances)) #o melhor é aquele cuja variância é maxima
            thresholds.append(supposed_th[supposed_variances.index(max(supposed_variances))])

    th_1 += 1

    #como a cada teste é testado o melhor, o "limiar global" melhor pode ser obtido pelo
    #máximo dos "melhores locais"
    return thresholds[variances.index(max(variances))], variances

```

A última função principal dessa implementação realiza atribuição de intensidade aos pixels da imagem de acordo com os limiares de corte passados por parâmetro.

```

def our_digitize(image, thresholds):
    regions = np.zeros(image.shape) #criando uma imagem vazia para armazenar o resultado

```

```

row, column = image.shape

for i in range(row):
    for j in range(column): #iterando sobre as linhas i e colunas j da imagem
        #iterando sobre os pontos de cortes (resultado da segmentação) e verificando
        #se a intensidade do pixel para atribuir a classe ao resultado
        for k in range(len(thresholds)):
            if image[i, j] <= thresholds[k]:
                regions[i, j] = k
                break
            elif image[i, j] > thresholds[-1]:
                regions[i, j] = len(thresholds)
                break

return regions

```

Dessa forma, após a execução da função do método de Otsu (de duas ou três classes) foi executada a função de atribuição de intensidades.

Executando o método próprio desenvolvido para segmentar a imagem com duas classes, observa-se que o limiar nesse caso foi bem semelhante ao obtido por meio do uso de uma função disponível na biblioteca (169): 171.

Para a função desenvolvida para o método de Otsu para 3 classes, os resultados obtidos foram bem precisos em comparação com aqueles obtidos pela biblioteca (156 e 218): 157 e 219.

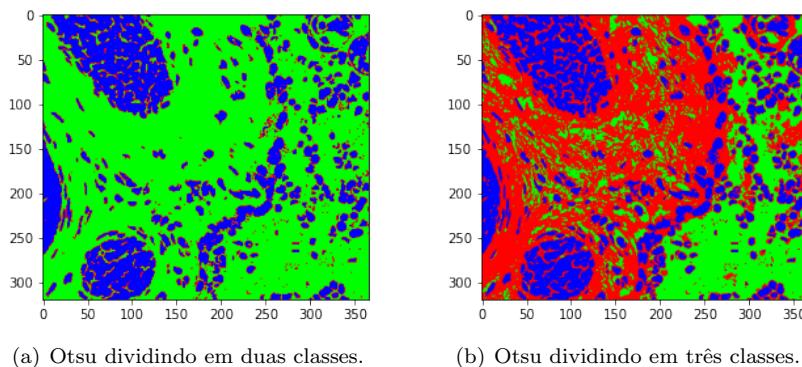


Figura 62: Aplicação de Otsu para a imagem da Figura do enunciado

## 7 Exercícios aula 10 - Morfologia Matemática

### 7.1 Exercício 4)

- 4) Construa um programa que receba a imagem A, forneça uma imagem limiarizada parecida com a indicada em B e aplique a operação binária necessária para obter um resultado similar ao indicado em C. Indique qual o valor de limiar escolhido e o elemento estruturante aplicado para obter C.

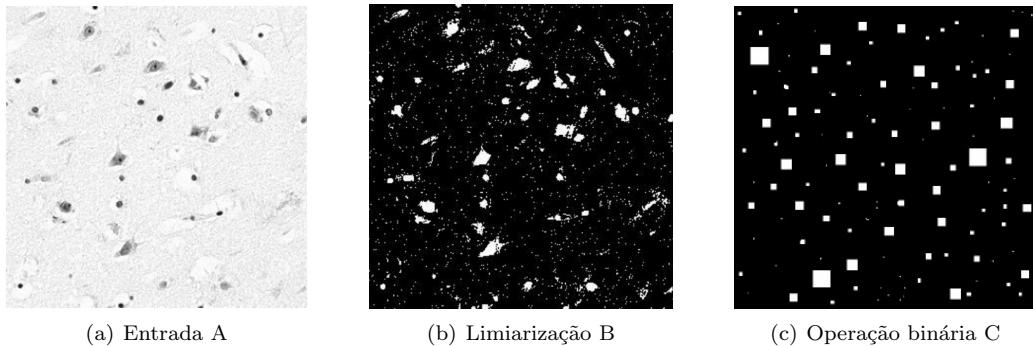


Figura 63: Imagens do enunciado.

Inicialmente foi criada uma função para realizar a limiarização/binarização de uma imagem em tons de cinza.

```
def thresholding(image, th):
    binary = np.zeros(image.shape) #criando a imagem resultado
    row, column = image.shape

    #iterando sobre os pixels da imagem em tons de cinza
    for i in range(row):
        for j in range(column):
            #se for maior ou igual que o limiar, recebe 0 (preto)
            if image[i, j] >= th:
                binary[i, j] = 0
            else: #se for menor, recebe 1 (branco)
                binary[i, j] = 1

    return binary
```

Aplicando a função para a imagem solicitada, obteve-se o resultado apresentado a seguir, com o limiar 223.

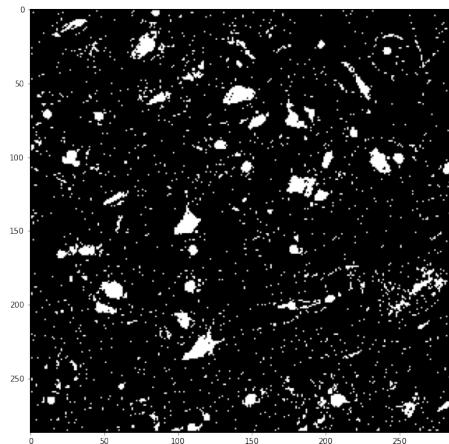


Figura 64: Imagem resultado da binarização (limiar 223).

Em seguida, foram testados diferentes elementos estruturantes para a transformação morfológica de abertura. Esse processo foi por métodos do pacote *skimage.morphology*, como: *square* e *opening*. Foi escolhida a transformação de abertura uma vez que o processo de interesse (B para C) tinha como objetivo eliminar ruídos brancos em uma imagem de fundo preto. O elemento estruturante **quadrado** foi escolhido já que as feições de interesse no resultado tinham formato semelhante à um quadrado.

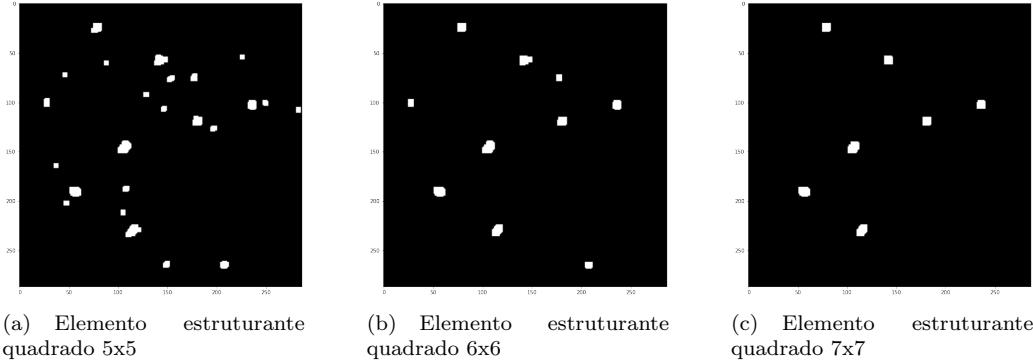


Figura 65: Resultado do processo de abertura pelo respectivo elemento estruturante.

Dessa forma, na imagem B, o valor de limiarização que mais se assemelhou a imagem resultante desse processo é 223. Já no caso da imagem C, a abertura com elemento estruturante no formato quadrado 6x6 permitiu um resultado mais próximo ao solicitado, apesar de possuir um elemento a mais. Isso se deve, provavelmente, às pequenas diferenças nos resultados da binarização ou divergências nos pacotes utilizados.

## 8 Exercícios aula 11 - Representação, descrição e análise de textura

### 8.1 Exercício 1)

1) Construa um código para receber imagens monocromáticas como entrada, 8 bits de quantização. O código deve ser capaz de fornecer os valores de:

- Haralick, com segundo momento angular, entropia e contraste. Use  $d=1$  e  $\theta=0$ ;
- Dimensão fractal (DF), usando *Box-counting*. A DF deve ser definida via coeficiente angular da regressão  $\log x \times \log y$ . Os dois primeiros valores parciais de DF, em função das iterações 1 e 2, também devem ser apresentados.

Os descritores devem ser organizados como vetores de características, respeitando a ordem posicional: momento angular, entropia, contraste, DF (coeficiente  $\log x \times \log y$ ) e DF iteração 1 e DF iteração 2. Apresente os vetores para cada imagem. As imagens são apresentadas nos próximos slides. Em seguida, observe os resultados numéricos e indique quais descritores apresentam as maiores diferenças para separar as imagens R0 de R3. Apresente os gráficos para ilustrar as posições espaciais dos descritores. Por exemplo, eixo x representa momento angular, eixo y a entropia e eixo z o contraste. Use a mesma estratégia para DF. Cada imagem é um ponto espacial em função das suas coordenadas/descritores.

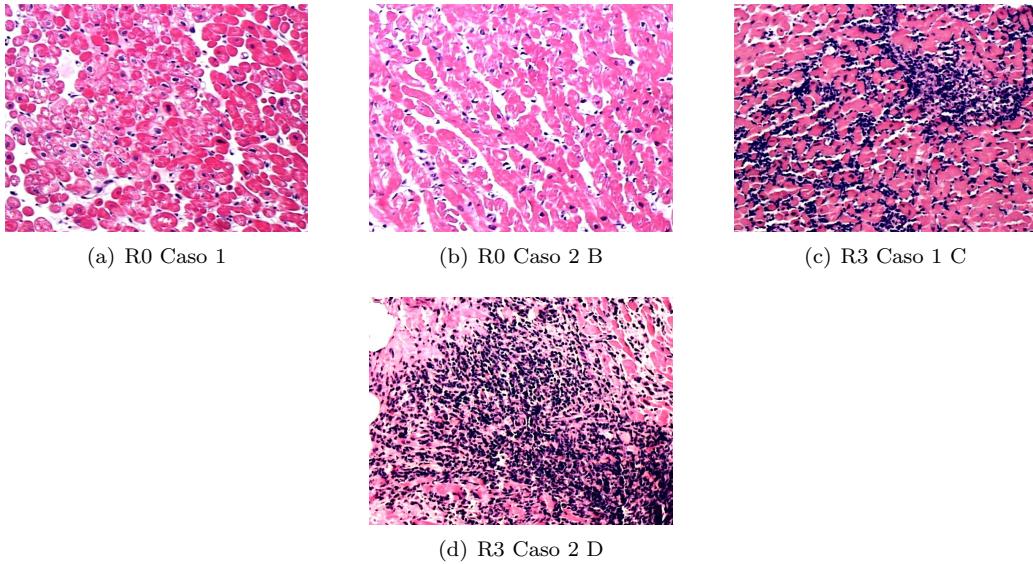


Figura 66: Imagens do enunciado.

Inicialmente foi criada uma função para realizar o cálculo da dimensão fractal extraindo os valores da iteração 1 e 2.

```

def fractal_dimension(Z, threshold=0.9):
    assert(len(Z.shape) == 2)
    # https://github.com/rougier/numpy-100
    def boxcount(Z, k):
        S = np.add.reduceat(
            np.add.reduceat(Z, np.arange(0, Z.shape[0], k), axis=0),
            np.arange(0, Z.shape[1], k), axis=1)

        # desconsiderando "boxes" vazias e totalmente escuras;
        return len(np.where((S > 0) & (S < k*k))[0])

    # Transformando a entrada em um array binário
    Z = (Z < threshold)

    # Buscando a menor dimensão da imagem
    p = min(Z.shape)

    # A menor potência de 2 menor ou igual a p
    n = 2**np.floor(np.log(p)/np.log(2))

    # Buscando o expoente
    n = int(np.log(n)/np.log(2))

    # Criando tamanhos de boxes sucessivas (2**n até 2**1)
    sizes = 2**np.arange(n, 1, -1)

    # Calculando a box de interação atual
    counts = []
    for size in sizes:
        counts.append(boxcount(Z, size))

```

```

# Calculando as 2 primeiras interações
divide1 = (np.log(counts[0])/np.log(2)**(1))
divide2 = (np.log(counts[1])/np.log(2)**(2))

# Definindo coeficientes angular e linear;
coeffs = np.polyfit(np.log(sizes), np.log(counts), 1)

#Ordem do array = DF, Interação 1, Interação 2

return -coeffs[0], divide1, divide2

```

Ao final da função os coeficientes angular e linear são calculados e os valores das 2 primeiras iterações são extraídos.

Após isso as imagens são carregadas e convertidas para "Grayscale" e prontamente tratadas para quantização em 8 bits.

```

...
gray_R0_1 = skimage.color.rgb2gray(rgb_R0_1)
gray_R0_1_reshaped = (gray_R0_1*256).astype('uint8')
...

```

É realizado a conversão da matriz grayscale para uma matriz de co-ocorrência (GLCM) e logo em seguida o cálculo dos desritores de Haralick e Dimensão Fractal com Box-Counting nas 4 imagens.

```

...
# Gerando matriz de co-ocorrência
GLCM_R01 = skimage.feature.graycomatrix(gray_R0_1_reshaped,[1],[0])

# Cálculo do Contraste
Contrast_Raw01_ = (skimage.feature.grycoprops(GLCM_R01,'contrast')).tolist()
Contrast_Raw01 = Contrast_Raw01_[0]
Contrast_R01 = Contrast_Raw01[0]

# Cálculo do segundo momento angular
ASM_Raw01_ = (skimage.feature.grycoprops(GLCM_R01,'ASM')).tolist()
ASM_Raw01 = ASM_Raw01_[0]
ASM_R01 = ASM_Raw01[0]

# Cálculo da Entropia
Entropy_R01 = np.mean(rank.entropy(gray_R0_1_reshaped, gray_R0_1_reshaped))

# Cálculo da Dimensão Fractal por Box Counting
DF_01 = fractal_dimension(gray_R0_1_reshaped)

Feature1 = (ASM_R01, Entropy_R01, Contrast_R01)
...

```

Um vetor de características é extraído ao final de cada cálculo (Feature1, 2, 3 e 4).

Os valores são apresentados para R0 (Casos 1 e 2):

Segundo Momento Angular R0 Caso 1 x Caso 2: 0.0004333937272793885, 0.000763825444787489

Entropia R0 Caso 1 x Caso 2: 7.558222211257797, 7.17843393273912

Contraste R0 Caso 1 x Caso 2: 456.13316645807254, 530.3034334584897

Dimensão Fractal R0 Caso 1 x Caso 2 (Com 1<sup>a</sup> e 2<sup>a</sup> interação): (1.0494211575090218, 2.0, 5.172007621563309), (1.1247795059721832, 2.0, 4.990904840502509)

R3 (Casos 1 e 2):

Segundo Momento Angular R3 Caso 1 x Caso 2: 0.00035044362573227663, 0.00040296302794011913

Entropia R3 Caso 1 x Caso 2: 7.624287853538409, 7.737200079237781

Contraste R3 Caso 1 x Caso 2: 427.13879849812264, 962.4073174801836

Dimensão Fractal R3 Caso 1 x Caso 2 (Com 1<sup>a</sup> e 2<sup>a</sup> interação): (0.6535452252659596, 2.0, 4.79252918868372), (0.8852501285416277, 2.0, 4.79252918868372)

E também são plotados para identificação de diferenças:

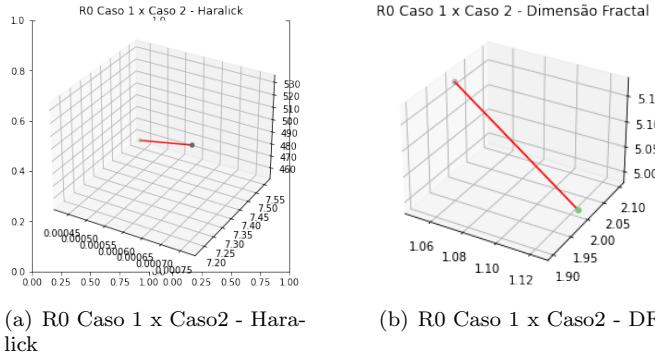


Figura 67: Comparativo Haralick - DF - R0.

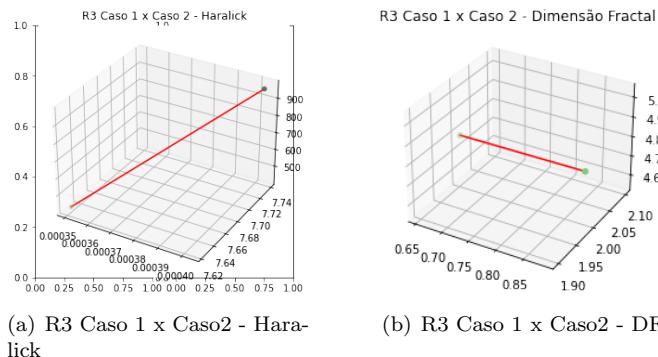


Figura 68: Comparativo Haralick - DF - R3.

Observa-se que os casos diferenciam-se entre os descritores, sendo que em R0 os descritores de Haralick apresentam uma diferença mais discreta quando em comparação com R3, invertendo-se esta proporcionalidade quando visto perante a dimensão fractal, onde em R0 a diferença se mostra mais significativa.