

**Universidade Estadual de Campinas**

Instituto de Computação

Introdução ao Processamento Digital de Imagens (MC920 / MO443)

Professor: Hélio Pedrini

## **Trabalho 2**

Julio Vinicius Amaral Oliveira – 230537

**Data de Entrega:** 28 de Abril de 2025

# Introdução

O seguinte trabalho visa demonstrar o uso das técnicas de meios-tonos e filtragem no domínio da frequência. Para o exercício 1, vamos usar um algoritmo de difusão de erro para quantizar os níveis de cinza de uma imagem, nosso intuito é preservar a percepção visual da imagem mesmo reduzindo a quantidade de tons de cinza dela. No segundo exercício, vamos utilizar a Transformada Rápida de Fourier para realizar operações como suavização, realce de bordas e compressão de dados.

## 1 Materiais e métodos

Para a resolução dos exercícios propostos, utilizamos as bibliotecas:

- OpenCV: Utilizada para ler, processar e exibir imagens.
- Numpy: Responsável pelo tratamento dos dados das imagens como arrays. É usada para realizar operações matemáticas, tanto as elementares como multiplicação e potenciação, quanto as mais complexas como a transformada de Fourier. Além de manipulação dos pixels de forma vetorizada, que é fundamental para o processamento de imagens de forma performática.
- matplotlib: Utilizado para a visualização das imagens no ambiente do Jupyter Notebook.

As atividades desenvolvidas foram organizadas em dois notebooks distintos, denominados `Parte1.ipynb` e `Parte2.ipynb`. No primeiro, abordamos as técnicas de meios-tonos com difusão de erro. No segundo, aplicamos filtragem no domínio da frequência através da Transformada Rápida de Fourier, implementando filtros passa-baixa, passa-alta, passa-faixa e rejeita-faixa, bem como procedimentos de compressão por limiarização de coeficientes.

## 2 Códigos auxiliares

Para a resolução dos desafios propostos, utilizamos um código auxiliar que serviu para encapsular a lógica de uma sequência de comandos. O código auxiliar criado foi:

```
1 def print_mono_image(image, title):  
2     plt.imshow(image, cmap='gray')  
3     if (title is not None):  
4         plt.title(title)  
5     plt.axis('off')  
6     plt.show()
```

Listing 1: Funções auxiliares para visualização de imagens

### 3 Exercícios e Discussão

#### 3.1 Exercício 1 – Técnicas de Meios-Tons com Difusão de Erro

Neste exercício, vamos aplicar técnicas de meios-tonos com difusão de erro em imagens digitais carregadas do diretório extt..//imagens/. Implementamos os métodos de Floyd-Steinberg, Burkes, Jarvis-Judice-Ninke, Sierra, Stevenson-Arce e Stucki. Cada algoritmo distribui o erro de quantização a pixels vizinhos segundo matrizes de pesos distintas.

##### Estruturas de Dados

As imagens foram representadas como `numpy.ndarray` bidimensionais. De modo a garantir precisão na propagação do erro, convertemos os valores de pixel para `float64`. Após aplicar a difusão, reconvertemos para `uint8` para gerar as imagens finais.

Adicionalmente, as máscaras de difusão de erro foram armazenadas em um dicionário Python (`error_diffusion_masks`), em que cada chave é o nome do método e cada valor é uma lista de tuplas (`i_rel`, `j_rel`, `peso`), por exemplo:

```
1 error_diffusion_masks = {  
2     "Floyd Steinberg": [  
3         (0, 1, 7/16),  
4         (1, -1, 3/16),  
5         (1, 0, 5/16),  
6         (1, 1, 1/16),  
7     ]  
8 }
```

Listing 2: Exemplo de definição das máscaras

Esse formato facilitou a criação de uma função genérica, usada tanto na varredura unidirecional quanto na serpentina, tornando o código muito mais elegante, pois para obter o resultado de todos os filtros, nós precisamos apenas iterar através de todas as máscaras.

##### Metodologia

Para implementar as técnicas de difusão de erro, desenvolvemos duas funções genéricas em Python, sendo que ambas recebem como parâmetros o caminho da imagem e a máscara de difusão (tuplas (`i_rel`, `j_rel`, `peso`)) extraída de `error_diffusion_masks`:

- **Leitura e preparo da imagem:**

1. Carregamos a imagem em escala de cinza

- **Varredura unidirecional (`apply_dithering_unidirectional`):**

1. Percorremos cada pixel  $(i, j)$  em ordem crescente de linhas e colunas.
  2. Binarizamos:  $\text{out\_image}[i, j] = 0$  se  $\text{image}[i, j] < 128$  senão 255.
  3. Calculamos o erro:  $\text{error} = \text{image}[i, j] - \text{out\_image}[i, j]$ .
  4. Para cada tupla da máscara, distribuímos  $\text{error} * \text{peso}$  a  $\text{image}[i+i_{\text{rel}}, j+j_{\text{rel}}]$ , caso as posições estivessem dentro dos limites da imagem.
- **Varredura em serpentina (apply\_serpentine\_scanning):**
    1. Alternamos a direção de varredura a cada linha: colunas esquerda→direita em linhas pares, direita→esquerda em linhas ímpares.
    2. Ajustamos o deslocamento horizontal do erro ( $j_{\text{rel}}$ ) invertendo seu sinal quando a linha é ímpar, para manter a matriz de difusão centrada corretamente.
    3. O restante do fluxo é idêntico ao método unidirecional (binarização, cálculo de erro, distribuição).
  - **Visualização e validação:**
    1. Após a execução de cada método, exibimos original, unidirectional e serpentine lado a lado via `matplotlib.pyplot.subplots`.
    2. Os resultados foram salvos em `../resultados/Parte 1/`.

## Resultados

**Floyd–Steinberg** Observamos que a difusão de erro se concentra imediatamente nos pixels vizinhos mais próximos, permitindo uma boa preservação dos detalhes da imagem original. Quando usamos varredura em serpentina, um efeito de linhas direcionais de uma cor que aparecem na varredura unidirecional é reduzido, minimizando a aparência das faixas.

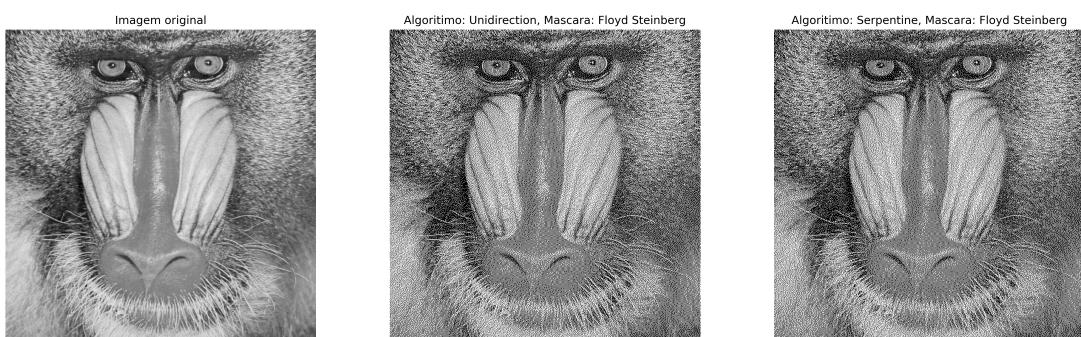


Figura 1: Resultado do método Floyd–Steinberg.

**Burkes** O método de Burkes distribui o erro em uma vizinhança maior, o que resulta em uma imagem geral mais com mais faixas da mesma cor, como podemos ver nitidamente na região do nariz do babuíno. As varreduras apresentaram um padrão muito próximo uma da outra, com a serpentina tirando um pouco dos detalhes em relação a varredura unidirecional, como é notório na região das bochechas do babuíno.

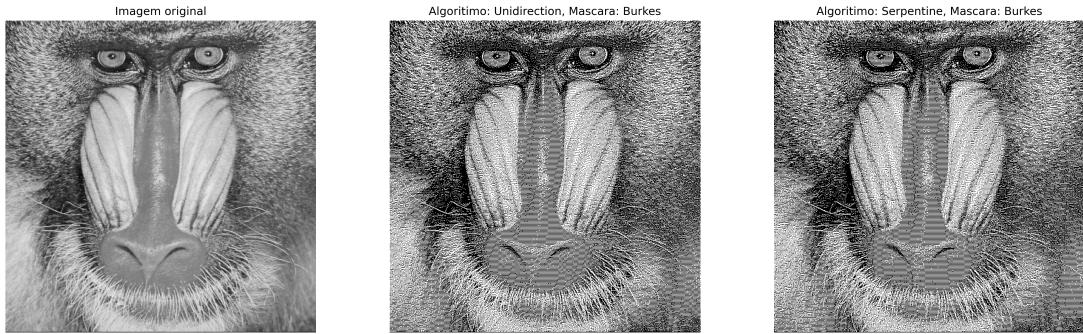


Figura 2: Resultado do método Burkes.

**Jarvis–Judice–Ninke** O método Jarvis–Judice–Ninke teve um dos melhores resultados dentre os métodos aplicados. Ele preservou muito bem os detalhes da imagem original, não foi apresentado áreas com concentração de cor, como no método de burkes. A varredura em serpentina, apresentou um resultado muito próximo a varredura unidirecional, mas tende a ter um padrão que é possível notar de forma mais fácil que há uma quantização das cores.

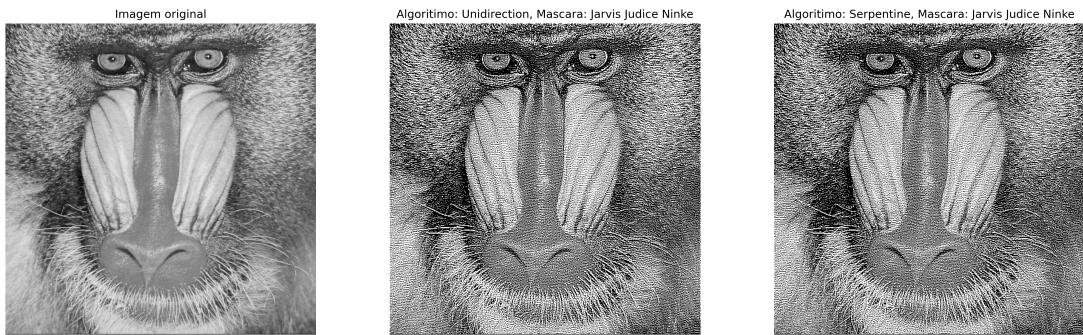


Figura 3: Resultado do método Jarvis–Judice–Ninke.

**Sierra** O algoritmo Sierra mostrou alguns padrões diagonais na imagem resultante, mas que preservaram muito bem a imagem original pois não apresentam áreas com grande concentração de cor. A varredura em serpentina apresentou uma imagem que aparenta ter um contraste maior do que a varredura unidirecional, tornando a imagem visualmente mais agradável.

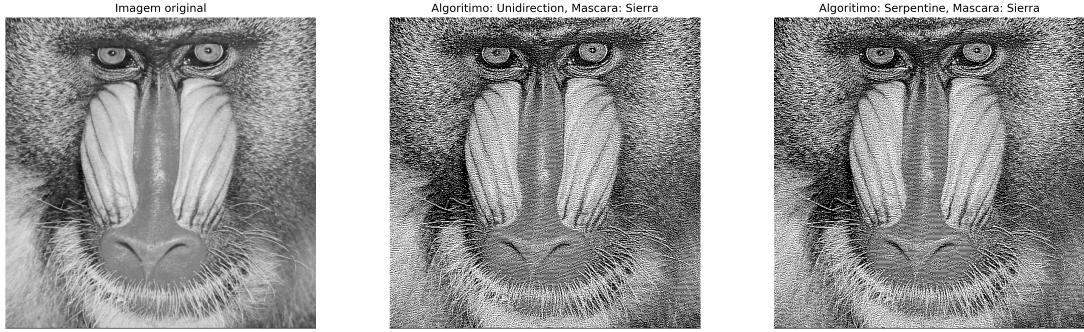


Figura 4: Resultado do método Sierra.

**Stevenson–Arce** O método Stevenson–Arce concentra a difusão de erro em uma vizinhança relativamente próxima ao pixel processado, fazendo com que haja uma região menor em que o erro é dissipado e criando assim, menos áreas de branco ou preto. É perceptível também que a varredura serpentina apresenta um padrão mais circular, enquanto a varredura unidirecional tende a apresentar um padrão mais direcional, como é possível ver bem na região do nariz do babuíno.

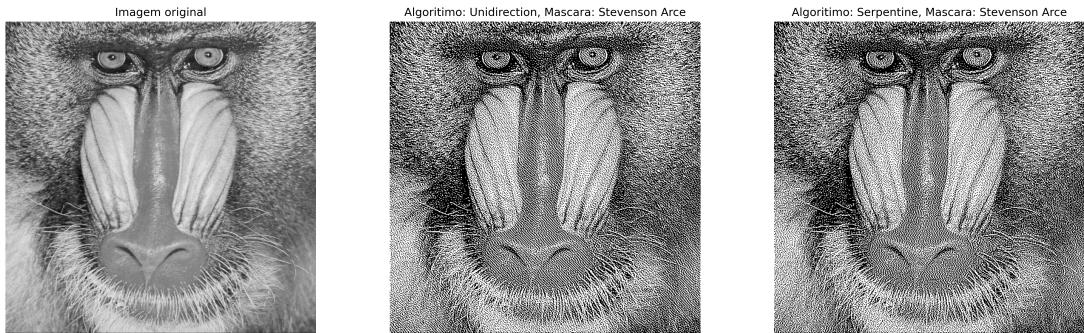


Figura 5: Resultado do método Stevenson–Arce.

**Stucki** O kernel Stucki estende mais o alcance da difusão. Isso gera um padrão de granulação mais "longo" com regiões mais longas de pixels branco e pixels escuros comparativamente ao método anterior. É notável também que a varredura serpentina removeu alguns detalhes da imagem, como as linhas da bochecha do babuíno, comparativamente com a varredura unidirecional.

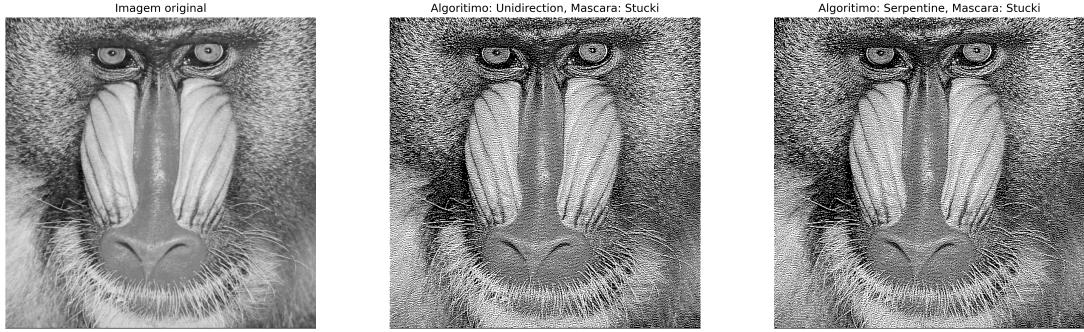


Figura 6: Resultado do método Stucki.

## 3.2 Exercício 2 – Filtragem no Domínio da Frequência

### Estruturas de Dados

Para organizar as máscaras de filtragem e os espectros intermediários, utilizamos principalmente dicionários em Python:

- `dict_filtros`: dicionário que mapeia cada nome de filtro (“passa-baixa”, “passa-alta”, “passa-faixa”, “rejeita-faixa”) a uma máscara. Cada máscara foi gerada calculando-se distâncias ao centro e comparando-se com raios definidos.
- `reconstructed_images`: dicionário cujas chaves correspondem aos nomes dos filtros e cujos valores são arrays `uint8` produzidos pela IFFT.

Dessa forma conseguimos escrever loops mais genéricos para processar cada chave dentro do loop, como:

```

1 for title , fft_spectrum in reconstructed_images.items () :
2     img_norm_uint8 = reconstruct_image(fft_spectrum)
3     print_mono_image(img_norm_uint8 , title )

```

### Metodologia

Para resolver o exercício adotamos o seguinte fluxo:

1. **Leitura e pré-processamento:** Carregamos a imagem em escala de cinza utilizando o OpenCV e a convertendo para `float64` para que tivéssemos precisão nas operações seguintes.
2. **Transformada de Fourier:** Aplicamos `np.fft.fft2` para obter o espectro bidimensional  $\mathcal{F}(u,v)$ , e então `np.fft.fftshift` para deslocar o componente DC para o centro da matriz.

**3. Construção das máscaras filtrantes:** Usamos `np.ogrid[:H,:W]` para gerar as matrizes de índices  $(y, x)$  e calculamos a distância

$d = \sqrt{(y - center\_row)^2 + (x - center\_column)^2}$ . Definimos quatro máscaras binárias do mesmo tamanho da imagem:

- `mask_pb`:  $d \leq r$  (passa-baixa)
- `mask_pa`:  $d \geq R$  (passa-alta)
- `mask_pf`:  $r \leq d \leq R$  (passa-faixa)
- `mask_rf`:  $d \leq r \vee d \geq R$  (rejeita-faixa)

**4. Aplicação dos filtros:** Para cada máscara, calculamos  $\text{FFT}_f = \text{fft_centered} \times \text{mask}$ . Armazenamos os espectros filtrados em variáveis que serão utilizadas para a reconstrução.

**5. Reconstrução ao domínio espacial:** Para cada espectro filtrado:

- (a) Revertemos ele com `np.fft.ifftshift`.
- (b) Aplicamos `np.fft.ifft2` e extraímos a parte real.
- (c) Obtemos seu valor absoluto e depois normalizamos linearmente para  $[0, 255]$  para no fim converter para `uint8`.

```

1 def reconstruct_image(fft_spectrum):
2     spectrum_shifted_pb=np.fft.ifftshift(fft_spectrum)
3     img_back=np.fft.ifft2(spectrum_shifted_pb)
4     img_real=np.real(img_back)
5     img_abs=np.abs(img_real)
6     img_norm=255*(img_abs-img_abs.min())/(img_abs.max()-img_abs.min())
7     img_norm_uint8=img_norm.astype(np.uint8)
8     return img_norm_uint8
9
10

```

Listing 3: Função de reconstrução de imagem

**6. Compressão por limiarização:** Para compressão, definimos um limiar  $T$  arbitrário e zeramos coeficientes:

$$\text{fft_comp} = \text{fft_centered}.copy(), \quad \text{fft_comp}[|\text{fft_comp}| < T] = 0.$$

Reconstruímos a imagem comprimida usando a mesma função de IFFT. A escolha do limiar foi feita com base na análise da figura, escolhemos um limiar que resultasse em uma figura semelhante a mostrada no PDF do exercício. Assim como a escolha de  $r$  e  $R$  que foram, respectivamente, os raios interno e externo do filtro passa-faixa.

## Resultados

**Espectro de Magnitude** Primeiro visualizamos o espectro de magnitude (escala logarítmica) da FFT centralizada, que exibe o componente DC no centro e mostra como as demais frequências se distribuem.

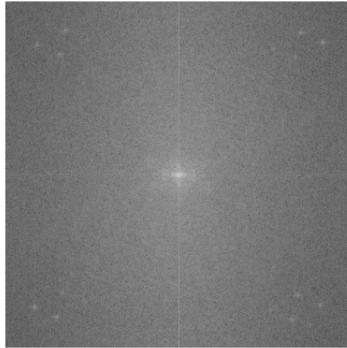


Figura 7: Magnitude do espectro centralizado.

**Filtro Passa–Baixa** O filtro passa–baixa preserva apenas as frequências baixas, resultando em um pequeno disco claro no centro do espectro. Na imagem reconstruída é perceptível que a imagem parece mais borrosa, causada pela perda de detalhes e uma suavização.

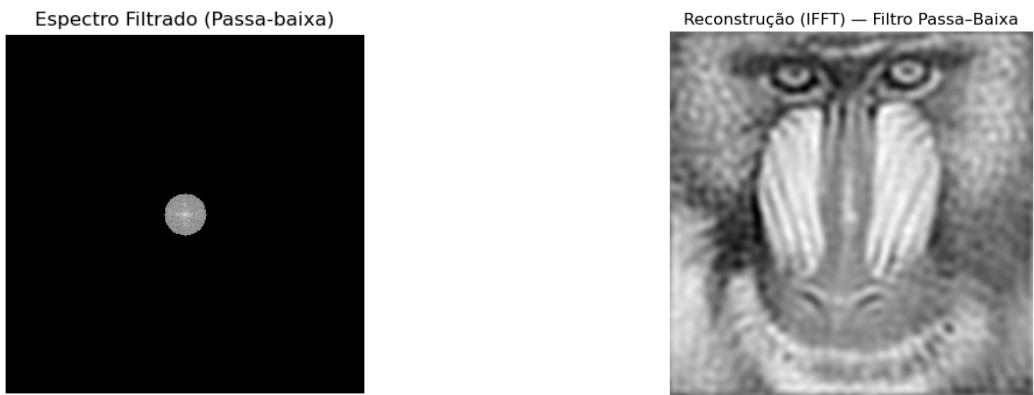
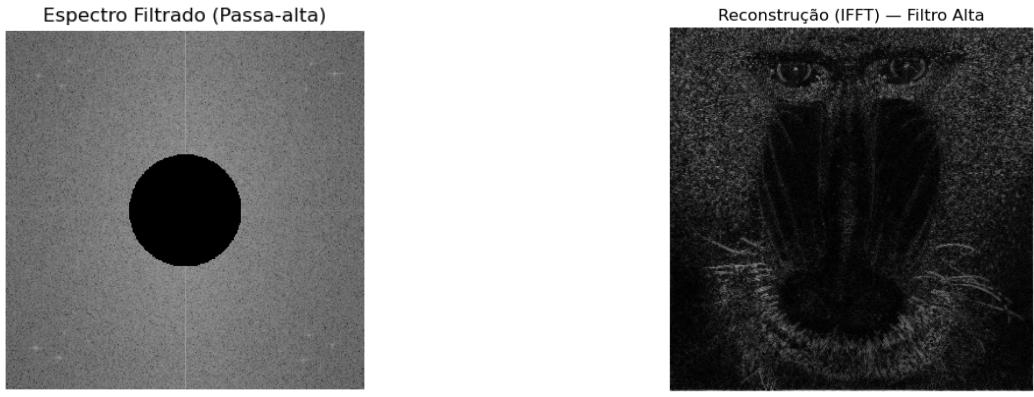


Figura 8: Espectro e reconstrução do filtro passa-baixa.

**Filtro Passa–Alta** O filtro passa–alta atenua as baixas frequências, destacando apenas as altas. No espectro, observa-se um grande disco preto central, e na imagem reconstruída predominam bordas e ruídos de alta frequência referente aos pelos do animal que tem uma coloração diferente do seu rosto.

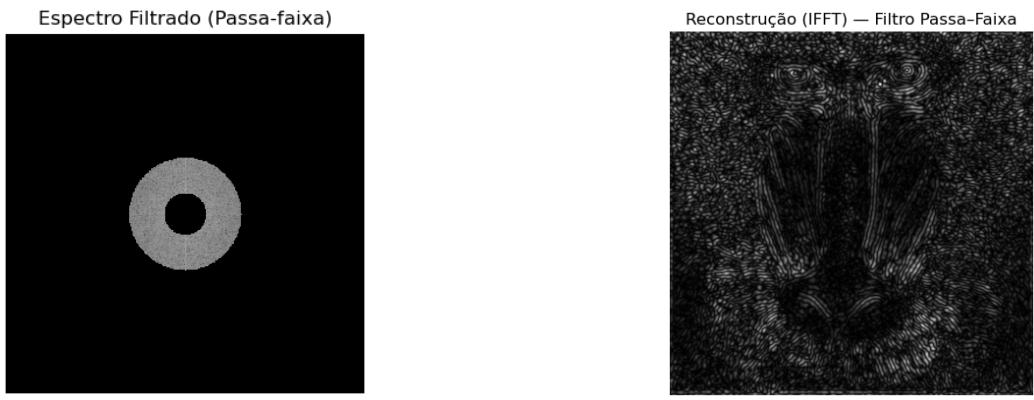


(a) Espectro filtrado (Passa–Alta).

(b) Reconstrução – Passa–Alta.

Figura 9: Espectro e reconstrução do filtro passa-alta.

**Filtro Passa–Faixa** O filtro passa–faixa preserva apenas uma faixa intermediária de frequências, resultando num anel claro no espectro. A imagem reconstruída mostra a combinação de suavização e realce.

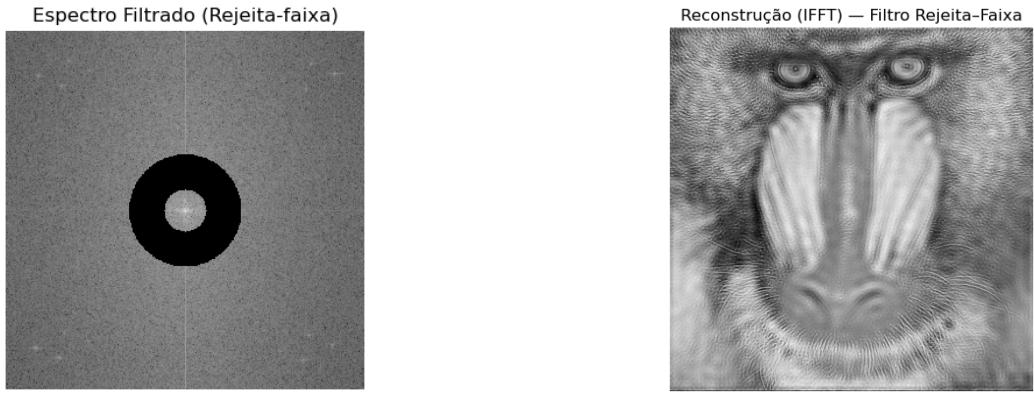


(a) Espectro filtrado (Passa–Faixa).

(b) Reconstrução – Passa–Faixa.

Figura 10: Espectro e reconstrução do filtro passa-faixa.

**Filtro Rejeita–Faixa** O filtro rejeita–faixa elimina frequências intermediárias, deixando apenas baixas e altas. No espectro, podemos ver um anel preto central com região clara interna, e a imagem reconstruída exibe simultaneamente suavização com a remoção de algumas bordas referentes ao pêlo do animal.



(a) Espectro filtrado (Rejeita–Faixa).

(b) Reconstrução – Rejeita–Faixa.

Figura 11: Espectro e reconstrução do filtro rejeita-faixa.

**Compressão por Limiar** Após aplicar limiar  $T$  e zerar coeficientes de baixa magnitude, a imagem comprimida apresenta redução significativa de detalhes, como mostrado abaixo:

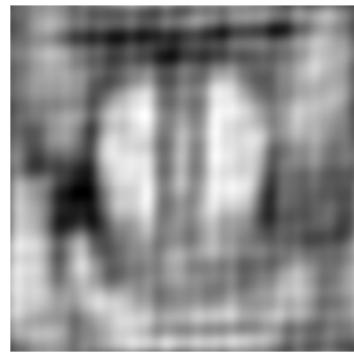


Figura 12: Imagem após compressão no domínio de frequência.

**Análise do Histograma Comparativo** Para quantificar o efeito da compressão por limiarização, comparamos os histogramas da imagem original e da comprimida:

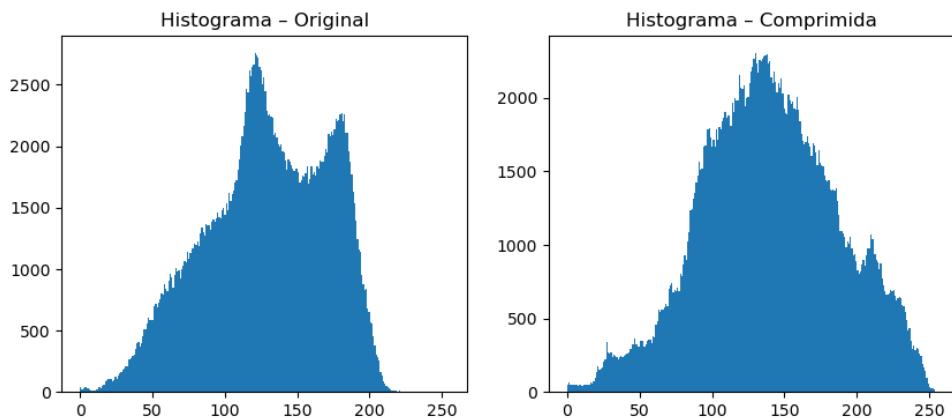


Figura 13: Histogramas antes e após compressão por limiar.

No histograma original observamos uma distribuição com dois picos claros em torno de níveis de cinza 120 e 180. Após a compressão, o histograma apresenta apenas um pico, mostrando que muitos coeficientes de baixa magnitude foram zerados. Esse “achatamento” da distribuição mostra que a compressão remove os detalhes tornando a representação mais uniforme.“

## 4 Conclusão

**Para o exercício 1** Conseguimos ver como a aplicação dos métodos de difusão de erro utilizando diferentes máscaras e varreduras impacta na qualidade da imagem. Por mais que todas as varreduras apresentem uma certa distorção quanto a imagem original, os resultados mostram que ambas as varreduras preservaram bem os detalhes da imagem original.

**Para o exercício 2** Conseguimos ver a aplicação de cada filtro no domínio da frequência, e como eles afetam a imagem original. Causando suavização, realce ou compressão. Além disso, realizar a compressão por limiarização mostrou como eliminar os coeficientes de baixa magnitude torna a imagem mais uniforme, causando perda de detalhes.