Processamento Digital de Imagens e Visão Computacional

Fundamentos, Algoritmos e Aplicações Integradas

AUTOR: FRANCISCO DE ASSIS ZAMPIROLLI

Processamento Digital de Imagens e Visão Computacional

Livro interativo com Python

Francisco de Assis Zampirolli

Sumário

PDI e VC — Python **12**
Organização 12

Prefácio **13**
Um livro vivo 13
Uso de ferramentas de Inteligência Artificial 13
Como este livro é produzido 13
A biblioteca `morph.py` 14
Exercícios de Programação (EPs) e validação automática 14
Estrutura e Validação 14
Idiomas e Conversão de Código 14
Código aberto 15
Antes de começar: Notebooks em Python 15

I Parte I — Fundamentos de PDI **16**

1 Fundamentos e Primeiros Passos **17**
1.1 Objetivos 17
1.2 Antes de começar: *Notebooks* em Python 17
1.3 Fundamentos 17
1.3.1 👁 Visão Computacional 18
1.3.2 🖼 Processamento Digital de Imagens (PDI) 18
1.3.3 Como as outras áreas se conectam 19
1.4 Etapas do PDI 19
1.5 Formação da Imagem e o Espectro 19
1.6 O que é uma Imagem Digital? 22
Representação Matemática 22
1.7 Configuração do Ambiente 22
1.7.1 📦 Bibliotecas Utilizadas 23
1.8 Fundamentos de Matrizes - atenção à cópia de referências 23
1.9 Lendo e Exibindo Imagens 24
Alternativa: Baixando a imagem localmente 25
1.10 Conversão de Tipos e Limiarização 26
Conversão para Tons de Cinza 26
Limiarização (*Thresholding*) 26
Exemplo prático de conversão e limiarização 27
1.11 Limiarização pelo método de Otsu 27
1.12 Acesso a Pixels 28
1.13 Resumo 30
1.14 🤖 Uso do NotebookLM como Tutor Complementar 30
Funcionalidades Disponíveis na Plataforma 30
1.15 Lista de Exercícios 31
Referências do Capítulo 32
1.16 💻 **Parte Prática com Exercícios de Programação** 32
🎯 Objetivo deste Caderno 32

§ Instruções Passo a Passo . . . 32
⚠️ Importante: Regras e Boas Práticas . . . 33
1.16.1 EP01_01 📏 Três métricas de distância em PDI . . . 34
1.16.2 EP01_02 📊 Desempenho Preditivo — Métricas de ML em VC . . . 41
1.16.3 EP01_03 ↗ *Mean Average Precision* (mAP) — Curva Precisão-Sensibilidade . . . 43
1.16.4 EP01_04 🖼 Leitura e Informações de uma Imagem em Matriz . . . 45
1.16.5 EP01_05 🔄 Negativo de uma Imagem em Tons de Cinza . . . 47
1.16.6 EP01_06 🎨 Conversão RGB → Tons de Cinza (ITU-R BT.601) . . . 49
1.16.7 EP01_07 ⚫ Limiarização Manual: Imagem Binária . . . 52
1.16.8 EP01_08 🎨 Remapeamento por Faixa de Intensidade . . . 53
1.16.9 EP01_09 🏁 Padrão de Tabuleiro: Matriz de Xadrez . . . 55
1.16.10 EP01_10 📄 Metadados: Leitura de Arquivo PGM . . . 56
1.16.11 EP01_11 ↗ Análise de Vizinhança: Filtro de Máximo 1D . . . 59

2 Da Captura ao Pixel - Amostragem, Quantização e Conectividade . . . 62
2.1 Objetivos . . . 62
2.2 O Olho e a Câmera - Elementos da Percepção Visual . . . 62
2.2.1 Visão humana . . . 62
2.2.2 Sensores digitais . . . 62
2.3 Ilusões de Ótica: Os Desafios da Percepção Visual . . . 63
2.3.1 Ambiguidade e Contexto . . . 63
2.3.2 Brilho e Contraste Local . . . 64
2.3.3 Geometria e Perspectiva . . . 64
2.4 O Modelo Matemático da Formação da Imagem . . . 64
2.5 Digitalização: Amostragem e Quantização . . . 65
2.5.1 Amostragem - Discretização do espaço . . . 65
2.5.2 Quantização - Discretização da intensidade . . . 65
2.5.3 Laboratório prático: Efeitos da amostragem e quantização . . . 65
2.5.4 Análise Técnica . . . 67
2.6 Relações entre Pixels - Topologia da Imagem . . . 69
2.6.1 Vizinhança . . . 69
2.6.2 Adjacência, conectividade e caminhos . . . 69
2.6.3 Distâncias entre pixels . . . 70
2.7 Armazenamento de Imagens . . . 70
2.7.1 Exemplo: Extração de Metadados e Localização GPS . . . 71
2.7.2 Por que esta separação é importante? . . . 73
2.8 Transformações Geométricas Básicas . . . 73
2.8.1 Translação . . . 73
2.8.2 Rotação . . . 74
2.8.3 Escala (Redimensionamento) . . . 75
2.8.4 Cisalhamento (*Shear*) . . . 76
2.9 Resumo . . . 77
2.10 🤖 Uso do NotebookLM como Tutor Complementar . . . 77
2.11 Lista de Exercícios . . . 78
Referências do Capítulo . . . 78
2.12 💻 **Parte Prática com Exercícios de Programação** . . . 78
🎯 Objetivo deste Caderno . . . 78
2.12.1 EP02_01 ⊙ Ajuste de Brilho e Contraste . . . 79
2.12.2 EP02_02 🔬 Subamostragem Espacial . . . 81
2.12.3 EP02_03 🎨 Quantização de Níveis de Cinza . . . 83
2.12.4 EP02_04 📐 Transformada de Distância em Imagem Binária . . . 84
2.12.5 EP02_05 Translação de Imagem . . . 87
2.12.6 EP02_06 🔄 Rotação de Imagem . . . 89
2.12.7 EP02_07 🔍 Redimensionamento (Escala) . . . 91

2.12.8 EP02_08 🔀 Cisalhamento (Shear) . . . 93
2.12.9 EP02_09 🧩 Transformação Afim Genérica . . . 95
2.12.10 EP02_10 🎯 Correção de Perspectiva (Homografia) . . . 97
2.12.11 EP02_11 🎯 Correção de Perspectiva em Imagem Real (Sudoku) . . . 98

3 Operações Espaciais: Intensidade, Histograma e Filtragem 102
3.1 Objetivos . . . 102
3.2 Operações em Nível de Intensidade . . . 102
3.2.1 Operações Aritméticas . . . 104
3.2.2 Mistura Ponderada (*Alpha Blending*) . . . 105
3.2.3 Operações Lógicas e Máscaras Bit a Bit . . . 107
3.3 Histograma de Imagens . . . 108
3.3.1 Equalização de Histograma . . . 109
3.3.2 Especificação de Histograma . . . 113
3.4 Fundamentos Espaciais: Vizinhança, Convolução e *Kernels* . . . 114
3.4.1 Vizinhança e Borda . . . 114
3.4.2 Correlação e Convolução . . . 114
3.4.3 O Papel do *Kernel* . . . 115
3.4.4 Exemplo Numérico: Correlação Passo a Passo . . . 116
3.5 Filtragem Espacial de Suavização . . . 118
3.5.1 Filtro de Média (*Box Filter*) . . . 118
3.5.2 Filtro Gaussiano . . . 119
3.6 Filtragem Espacial de Realce . . . 120
3.6.1 Laplaciano . . . 121
3.6.2 Operador de Sobel . . . 123
3.6.3 *Unsharp Masking* (USM) . . . 125
3.7 Filtros de Ordem: Filtro da Mediana . . . 127
3.7.1 Ruído Impulsivo: Sal e Pimenta . . . 127
3.7.2 Filtro da Mediana . . . 128
3.8 Aplicação Prática: Pré-processamento para Segmentação . . . 130
3.9 Resumo . . . 131
3.10 🤖 Uso do NotebookLM como Tutor Complementar . . . 132
3.11 Lista de Exercícios . . . 132
Referências do Capítulo . . . 133
3.12 💻 **Parte Prática com Exercícios de Programação** . . . 133
🎯 Objetivo deste Caderno . . . 133

4 Morfologia Matemática e Segmentação de Imagens 136
4.1 Objetivos . . . 136
4.2 Limiarização . . . 137
4.2.1 Imagem de Moedas . . . 138
4.2.2 Escolha do Pré-processamento para o Otsu . . . 138
4.2.3 Resultado: CLAHE como Melhor Pré-processamento . . . 141
4.3 Morfologia Matemática . . . 142
4.3.1 Erosão e Dilatação . . . 142
4.3.2 Abertura e Fechamento . . . 144
4.3.3 Reconstrução Morfológica . . . 147
4.3.4 Preenchimento de Buracos e Remoção de Bordas . . . 149
4.3.5 Pipeline de Limpeza Binária com CLAHE . . . 151
4.3.6 Morfologia em Tons de Cinza . . . 152
4.4 Segmentação de Imagens: Fundamentação e Taxonomia . . . 154
4.4.1 Rotulação . . . 155
4.4.2 Transformada de Distância . . . 158
4.4.3 Segmentação por *Watershed* . . . 161

4.5 Extração de Componentes e Descritores de Forma . . . 163
4.5.1 connectedComponentsWithStats . . . 163
4.5.2 Descritores de Forma . . . 166
4.5.3 Conexão com Detecção de Objetos Moderna . . . 166
4.5.4 Segmentação por *K-Means* . . . 168
4.6 Aplicação Prática: Pipeline Completo de Contagem . . . 169
4.7 Resumo . . . 169
4.8 🤖 Uso do NotebookLM como Tutor Complementar . . . 169
4.9 Lista de Exercícios . . . 170
Referências do Capítulo . . . 170
4.10 💻 **Parte Prática com Exercícios de Programação** . . . 170
🎯 Objetivo deste Caderno . . . 171

5 Transformadas e Compressão **173**
5.1 Objetivos . . . 173
5.2 Configuração do Ambiente . . . 173
5.3 Transformada de Fourier Discreta 2D . . . 174
5.3.1 Fenômeno: o que uma imagem “parece” no domínio da frequência? . . . 174
5.3.2 O Experimento da Grade: Construindo a Imagem do Zero . . . 175
5.3.3 Definição Matemática . . . 176
5.3.4 As Funções de Base da DCT . . . 176
5.3.5 Implementação e Visualização . . . 176
5.3.6 O que a Magnitude e a Fase carregam? . . . 176
5.3.7 Simulador: Reconstruindo Sinais com Senoides . . . 178
5.4 Teorema da Convolução . . . 178
5.4.1 Eficiência Computacional: DFT vs Convolução Direta . . . 180
5.5 Filtros no Domínio da Frequência . . . 183
5.5.1 Filtros Passa-Baixa . . . 184
5.5.2 Filtros Passa-Alta e Passa-Banda . . . 185
5.5.3 Remoção de Ruído Periódico . . . 188
📌 Síntese — Filtros Espectrais . . . 189
5.6 *Wavelets* e Multirresolução . . . 190
5.6.1 O Limite de Fourier: Onde ocorreu o evento? . . . 190
5.6.2 Transformada Wavelet Discreta 2D . . . 190
5.6.3 Famílias de Wavelets . . . 191
5.6.4 📌 Síntese - Fourier vs *Wavelets*: quando usar cada uma? . . . 195
5.7 Compressão de Imagens . . . 196
5.7.1 Taxonomia das Redundâncias . . . 196
5.7.2 Transformada de Cossenos Discreta (DCT) . . . 196
5.7.3 Quantização: a principal fonte de compressão . . . 197
5.7.4 Pipeline JPEG . . . 199
5.7.5 Simulador Interativo: Quantização DCT . . . 201
5.8 Comparação de Formatos de Imagem . . . 201
5.8.1 Características dos Formatos . . . 201
5.8.2 Métricas de Qualidade . . . 203
5.8.3 Inspeção Visual: Qualidade não é só PSNR . . . 203
📌 Síntese — Compressão JPEG . . . 207
5.9 Aplicação Prática: Remoção de Ruído por Filtragem Espectral . . . 208
5.10 Resumo do Capítulo . . . 208
🔑 Fundamentos Essenciais . . . 208
5.11 🤖 Uso do NotebookLM como Tutor Complementar . . . 209
5.12 Lista de Exercícios . . . 209
Referências do Capítulo . . . 209

Referências **211**

Lista de Figuras

1.1 Diagrama relacional detalhando as distinções fundamentais, sinergias e interconexões entre Processamento de Imagens e Visão Computacional no contexto de sistemas baseados em imagens. 18
1.2 Representação do fluxo sequencial de processamento: a saída de cada nível torna-se a entrada do nível subsequente. 20
1.3 Fluxo detalhado do PDI: da aquisição sensorial à extração de atributos e reconhecimento automatizado, incluindo em processamento colorido e compressão. 20
1.4 Representação do espectro visível e sua posição em relação às demais radiações eletromagnéticas, destacando a variação dos comprimentos de onda de 380 nm a 750 nm. 21
1.5 (A) Espectro eletromagnético completo em escala logarítmica, com destaque para a faixa visível. (B) Detalhe da luz visível (380-750 nm) e suas cores componentes. (C) Decomposição da luz branca pelo prisma: menor comprimento de onda sofre maior refração, separando UV, visível e infravermelho. 21
1.6 Exemplo de imagem sintética gerada por matriz aleatória 4x6. 24
1.7 Mandrill (Mandrillus sphinx) in Gabon GPS: (-1.93867, 13.09674). Crédito: Julien Renoult (CC BY 4.0). 25
1.8 Processamento básico de imagens: (a) imagem original, (b) imagem em tons de cinza, (c) imagem binarizada por limiar (T=128). 27
1.9 Comparação entre limiarização manual (T=128) e automática (Otsu) sobre a imagem em tons de cinza. 28
1.10 Exemplo de matriz sintética 5×5 com pixel central branco. 29
1.11 Estúdio do NotebookLM. 31
1.12 Simulador: Distâncias Euclidiana, *City-block* e *Chessboard* 36
1.13 Simulador: Desempenho Preditivo — Métricas de ML 43
1.14 Simulador: *Mean Average Precision* (mAP) — Curva P-S 46
1.15 Simulador: Estatísticas de Pixels 48
1.16 Simulador: Transformação de Negativo 50
1.17 Simulador: Percepção de Cor e Pesos ITU-R 51
1.18 Simulador: Limiarização Interativa 53
1.19 Simulador: Ajuste de Brilho e Contraste 55
1.20 Simulador: Gerador de Xadrez 57
1.21 Simulador: Estrutura do Arquivo PGM 59
1.22 Simulador: Filtro de Máximo Local 1D 61

2.1 Comparativo didático entre o sistema visual biológico e o eletrônico: no topo, a anatomia do olho humano destacando a retina e os fotorreceptores (cones e bastonetes); abaixo, a estrutura de uma câmera digital detalhando o sensor CMOS, a matriz de filtros de Bayer (RGGB) e o processo de quantização digital realizado pelo ADC. 63
2.2 Coletânea de desafios perceptivos: (topo esquerdo) Vaso de Rubin — ambiguidade figura-fundo; (topo central) Elefante de Shepard — incongruência geométrica; (topo direita) Sombra de Adelson — constância de brilho baseada no contexto; (inferior esquerdo) Grade de pontos — inibição lateral; (inferior direito) Escada de Schröder. 64
2.3 Efeito da subamostragem. Os títulos exibem as dimensões (W x H) e o tamanho da matriz em memória (KB). 67
2.4 Efeito da redução da profundidade de bits. Os títulos exibem a quantidade de bits, níveis e o tamanho em memória (KB). 68
2.5 Ilustração de vizinhanças 4 em uma matriz 3x3. No centro (1,1), o pixel de interesse. 69

2.6 APA Quilombos do Médio Ribeira - Barbudo-rajado (Malacoptila striata). Crédito: Thomas Fuhrmann (CC BY-SA 4.0). 72
2.7 Exemplo de translação da imagem do pássaro com deslocamentos (50,50) e (100,50). 74
2.8 Exemplo de rotação da imagem do pássaro em 30° e 45° usando interpolação bilinear. 75
2.9 Comparação de interpolação com zoom no detalhe do olho (recorte 60×60, ampliado 4×). Note o efeito de blocos no vizinho mais próximo vs. a suavização na bilinear. 76
2.10 Exemplo de cisalhamento da imagem do pássaro: horizontal (shx=0.3), vertical (shy=0.3) e combinado (shx=0.2, shy=0.2). 77
2.11 Simulador: Ajuste de Brilho e Contraste 80
2.12 Simulador: Subamostragem 82
2.13 Simulador: Quantização 84
2.14 Simulador: Ajuste de Brilho e Contraste 86
2.15 Simulador: Translação (tx, ty) 88
2.16 Simulador: Rotação (ângulo) 90
2.17 Simulador: Redimensionamento (Nearest vs Bilinear) 92
2.18 Simulador: Cisalhamento (shx, shy) 94
2.19 Simulador: Transformação Afim (matriz 2×3) 96
2.20 Simulador: Correção de Perspectiva (Homografia) 98
2.21 Aquisição da imagem de um Sudoku à esquerda. À direita, conversão para tons de cinza e redimensionamento. Crédito: Héctor Rodríguez de Guardamar, Espanha (CC BY 2.0). 100
2.22 Simulador: correção de perspectiva do Sudoku. 101

3.1 *Mandrill* (*Mandrillus sphinx*) fotografado em ambiente natural na África do Sul. A imagem possui resolução de 500 px e contém metadados EXIF com coordenadas GPS aproximadas (-26.20473, 28.22662). Crédito: Carlos Guilherme Rodrigues (CC BY-SA 3.0). 104
3.2 Operações aritméticas saturadas: adição de constante (clareamento) e subtração de constante (escurecimento com saturação em 0). 105
3.3 Retrato de um leopardo (*Panthera pardus*) em ambiente natural. Crédito: C. Brück (CC BY-SA 4.0). 106
3.4 *Alpha blending* entre recortes alinhados de mandrill e do leopardo (Figure 3.3) para diferentes valores de . Em =1 vê-se apenas mandrill; em =0, apenas o leopardo; valores intermediários fundem os olhares das duas imagens proporcionalmente. 107
3.5 Operações lógicas bit a bit com máscara circular: NOT (negativo), AND (isolamento da ROI) e OR (iluminação da ROI). 108
3.6 Histogramas da imagem original, de uma versão escurecida e de uma versão clareada. Os valores entre parênteses indicam a quantidade de pixels saturados em 0 (preto) e 255 (branco). 109
3.7 Equalização de histograma em imagem 5×5 com 3 bits (L=8): imagem original com tons concentrados em {2,3,4}, imagem equalizada com tons redistribuídos para {1,5,7}, e os respectivos histogramas evidenciando o espalhamento das frequências. 112
3.8 Equalização de histograma: mm.equalize (global via CDF) vs. CLAHE (adaptativa com limitação de contraste). Os histogramas revelam o espalhamento progressivo das intensidades. 113
3.9 Especificação de histograma: mandril mapeada para o perfil tonal do leopardo. A CDF da saída aproxima a CDF de referência. 114
3.10 Correlação com kernel de média 3×3: versão didática (mm.conv0) vs. vetorizada (mm.conv via cv2.filter2D). A diferença máxima de 27 ocorre nas bordas, onde o tratamento de padding difere. 116
3.11 Patch 5×5 extraído da imagem do mandrill (posição [250:255, 250:255]). A janela amarela destaca a vizinhança 3×3 centrada no pixel [1,1] onde a correlação será calculada. 117
3.12 Filtro de média com kernels de tamanho crescente (3×3, 7×7, 15×15). O borramento das bordas aumenta com o tamanho do kernel. 118
3.13 Kernel Gaussiano 5×5 (=1): pesos normalizados, maiores no centro e decrescentes radialmente. A grade facilita a leitura de cada coeficiente. 119
3.14 Comparação entre filtro de média e Gaussiano (janela 9×9, =0 para cálculo automático). O Gaussiano preserva melhor as bordas, visível no detalhe do rosto. 120

3.15 . . . 122
3.16 Laplaciano aplicado à imagem do mandrill: resposta bruta (bordas detectadas) com w4 e w8, e imagens realçadas pela subtração do Laplaciano. w8 é mais sensível às diagonais. . . . 123
3.17 Operador de Sobel na imagem mandrill: Gx detecta bordas verticais, Gy detecta bordas horizontais, e a magnitude | f| combina ambos, revelando todas as bordas independente de direção. . . . 125
3.18 Unsharp Masking na imagem do mandrill com =1 e k variando de 0.5 a 3.0. Para k>2 surgem artefatos nas bordas (halos) e o ruído de fundo começa a ser visível. . . . 126
3.19 Ruído sal e pimenta com densidades crescentes (2%, 5%, 10%). O parâmetro prob indica a fração total de pixels corrompidos, metade sal (255) e metade pimenta (0). . . . 128
3.20 Comparação de filtros para remoção de ruído sal e pimenta (10%): Gaussiano, Média, Mediana, Bilateral e Morfológico. Linha superior: imagens completas; linha inferior: crop da região de interesse. . . . 130
3.21 Pipeline de pré-processamento: CLAHE → Gaussiano → Canny. Cada etapa prepara a imagem para a seguinte, resultando em bordas limpas e bem definidas. . . . 131
3.22 Simulador: Distâncias Euclidiana, *City-block* e *Chessboard* . . . 134

4.1 Imagem com moedas de vários tipos. Crédito: GAZI.MD.AHAD (CC BY-SA 4.0). . . . 138
4.2 Comparação dos pré-processamentos: imagem | histograma+T* | ²B(T) | Otsu. A melhor separação bimodal indica o limiar mais confiável. . . . 141
4.3 Erosão e dilatação em imagem binária 10×10 com elemento estruturante 'L' assimétrico 3×3. Validação da dualidade erosão–dilatação. . . . 144
4.4 Simulador interativo de erosão morfológica: visualização do critério de inclusão do elemento estruturante assimétrico B_L sobre a imagem binária A. . . . 145
4.5 Abertura, fechamento, composição e filtro sequencial alternado aplicados à binarização Otsu das moedas com CLAHE. Elemento estruturante: disco 13×13. . . . 147
4.6 Elemento estruturante em formato de cruz (B_{cruz}) utilizado para conectividade-4. . . . 148
4.7 Pipeline de Reconstrução Morfológica por Dilatação Condicionada: a máscara contém dois objetos, o marcador isola apenas o núcleo do objeto principal, e as iterações subsequentes em laço reconstroem sua forma exata até a convergência. . . . 149
4.8 Pipeline de preenchimento de buracos (*clohole*): o marcador é gerado restringindo o frame da borda ao complemento f^c. As iterações de dilatação condicionada preenchem o fundo externo e o operador final preserva unicamente os buracos internos isolados. . . . 150
4.9 Pipeline de eliminação de estruturas de borda (*edgeoff*): o marcador captura as raízes conectadas às extremidades da matriz, a reconstrução delimita a extensão desses elementos e a subtração booleana preserva unicamente os objetos totalmente internos. . . . 151
4.10 Pipeline completo de segmentação com CLAHE: Otsu → fechamento → clohole → abertura (remoção de ruído) → edgeoff. . . . 152
4.11 Morfologia em tons de cinza emparelhada com seus respectivos histogramas: imagem seguida por sua distribuição de frequências de intensidade. Elemento estruturante: disco de raio 19. . . 154
4.12 Simulador interativo de rotulação de componentes conexas (*connected component labeling*): visualização da expansão flood-fill, conectividade-4 e conectividade-8. . . . 157
4.13 Efeito da conectividade na rotulação: a imagem binária 10×10 contém pixels diagonalmente adjacentes. Com conectividade-4, esses pixels formam componentes distintas; com conectividade-8, fundem-se em uma única componente. . . . 158
4.14 Simulador interativo da Transformada de Distância (TD) iterativa via erosão em tons de cinza. Pixels fora da imagem agora assumem o valor máximo (144), propagando os custos apenas a partir do fundo interno. . . . 159
4.15 Transformada de Distância em imagem binária 10×10. Esquerda: imagem original (foreground = 255). Centro: mm.dist1 iterativa (erosões com elemento cruz), exibindo o número de erosões sobrevividas por cada pixel. Direita: mm.dist (L2 Euclidiana). Os valores internos confirmam a equivalência entre as duas abordagens na métrica induzida pelo elemento estruturante. . . . 160

4.16 Simulador interativo do Algoritmo Watershed por propagação morfológica. Desenhe a máscara, posicione os marcadores e ajuste o Elemento Estruturante para observar a inundação. Quando as bacias se encontram simultaneamente, o empate é resolvido assumindo uma das regiões de forma aleatória. 162
4.17 Pipeline watershed (cv2) em imagem binária 20×20 com dois discos sobrepostos: transformada de distância, marcadores por threshold e resultado final. 163
4.18 164
4.19 165
4.20 166
4.21 167
4.22 168
4.23 168
4.24 Simulador: Distâncias Euclidiana, *City-block* e *Chessboard* 172

5.1 Decomposição de Fourier 1D: uma onda quadrada (linha tracejada) é aproximada pela soma das primeiras senoides (linhas coloridas). Quanto mais termos, melhor a aproximação. 174
5.2 Toda frequência no espectro (ponto isolado) corresponde a uma onda senoidal 2D rotacionada no domínio espacial. 175
5.3 Diagrama conceitual do espectro de Fourier 2D centrado. 177
5.4 Experimento de troca de fase: Imagem A (moedas) e Imagem B (padrão geométrico) reconstruídas com magnitudes e fases trocadas. O resultado confirma que a estrutura visual segue a fase — a imagem reconstruída com a fase de B se parece com B, independentemente de qual magnitude foi usada. 178
5.5 Simulador interativo da decomposição de Fourier 1D: visualização da soma de senoides com diferentes frequências, amplitudes e fases. Adicione termos e observe a convergência para formas de onda arbitrárias. 179
5.6 Pipeline completo: Butterworth + Notch para remoção de ruído misto. 181
5.7 Sem padding, um deslocamento severo faz a imagem vazar para o lado oposto (convolução circular). 182
5.8 Verificação do Teorema da Convolução: a diferença pixel a pixel entre a convolução espacial (cv2.filter2D) e a multiplicação em frequência (FFT) é numericamente nula — confirmando a equivalência teórica. 183
5.9 A Dualidade Perigosa: O corte abrupto na Frequência (Cilindro) transforma-se obrigatoriamente numa Sinc espacial. Suas ondulações causam o *ringing* fantasma nas bordas da imagem. 184
5.10 Filtros passa-baixa. 185
5.11 Simulador interativo de filtros no domínio da frequência. 186
5.12 Comparação entre filtros passa-baixa: Ideal (D =30), Gaussiano (D =30) e Butterworth (D =30, n=2). Perfis de H(u,v) ao longo de uma linha central e imagens filtradas correspondentes. . . . 187
5.13 Filtro passa-alta Gaussiano. (a) Original; (b) Filtro passa-alta (D =30) - as bordas das moedas e fundo texturizado são realçados. 188
5.14 Remoção de ruído periódico via filtro notch no domínio da frequência: (a) imagem com ruído senoidal, (b) espectro mostrando os picos do ruído, (c) máscara notch centrada nos picos, (d) imagem restaurada. 189
5.15 Fourier global é cego para a posição. Os espectros não dizem onde as bordas estão. 190
5.16 Funções da Wavelet ( ). Note como elas rapidamente decaem para zero (suporte compacto), ao contrário das senoides infinitas de Fourier. 192
5.17 Diagrama da decomposição wavelet 2D em dois níveis. 192
5.18 Decomposição wavelet 2D de 2 níveis com wavelet Haar: subbandas LL, LH, HL, HH em cada nível. As subbandas de detalhe revelam estruturas orientadas em diferentes escalas. 193
5.19 Comparação entre famílias de wavelets: Haar, db4, sym4 e bior2.2. Subbanda LL (aproximação) e HH (diagonal) para cada escolha, ilustrando o compromisso entre compactação e suavidade. 194

5.20 Reconstrução wavelet com limiamento de coeficientes (hard thresholding): à medida que o limiar aumenta, mais detalhes são zerificados, produzindo imagens progressivamente mais suaves. Métrica PSNR quantifica a perda de qualidade. 195
5.21 O Alfabeto Visual do JPEG: As 64 funções de base da DCT-II. O coeficiente DC fica no topo esquerdo (suave). Ao descer e avançar à direita, a oscilação espacial aumenta drasticamente. . 197
5.22 DCT 2D em bloco 8×8: coeficientes e reconstrução progressiva. 199
5.23 Pipeline JPEG simplificado: DCT em blocos 8×8, quantização com diferentes fatores de qualidade e reconstrução via IDCT. O artefato de blocos torna-se visível para qualidades baixas (Q=10–20). 201
5.24 Simulador interativo de compressão DCT-JPEG: ajuste o fator de qualidade e visualize em tempo real os coeficientes zerados, o bloco reconstruído e o erro de quantização. 202
5.25 Forte compressão (Q=10). À esquerda o artefato de bloco clássico da DCT. À direita, a suavização de bordas característica do WebP. 204
5.26 Curva taxa-distorção: PSNR vs tamanho de arquivo para JPEG, WebP e PNG. 205
5.27 Mapas de erro JPEG em diferentes qualidades: artefatos de bloco nas bordas. 207
5.28 Mapa conceitual do domínio da frequência. 208

Lista de Tabelas

1.2 Principais tipos de imagem digital e seus respectivos conjuntos de valores possíveis para cada pixel. . . . 22

2.1 Comparativo entre estratégias de compactação e eficiência de processamento. . . . 69
2.2 Comparativo de métricas de distância aplicadas à malha de pixels. . . . 70
2.3 Principais formatos de armazenamento e suas aplicações em PDI. . . . 70
2.4 Métodos de interpolação disponíveis em `mm.resize` e seus respectivos números de vizinhos utilizados no cálculo. . . . 75

3.1 Algoritmo de equalização de histograma. . . . 109
3.2 Interpretação dos coeficientes do *kernel*. . . . 115
3.3 Etapas do *Unsharp Masking*. . . . 125
3.4 Média vs. mediana com um pixel corrompido. A mediana ignora o outlier; a média é deslocada ~40 níveis. . . . 127

4.2 Propriedades de abertura e fechamento. . . . 146
4.3 Sequências do filtro sequencial alternado `mm.asf`. . . . 146
4.5 Taxonomia dos métodos de segmentação baseados em propriedades de homogeneidade e conectividade espacial. . . . 155
4.6 Pipeline morfológico típico do *watershed* baseado em marcadores. . . . 161

5.2 Comparação de complexidade computacional entre convolução direta e filtragem via FFT. . . . 180
5.7 Tipos de redundância em imagens e como são exploradas. . . . 196
5.9 Comparação entre os principais formatos de imagem rasterizados. . . . 203

PDI e VC — Python

Bem-vindo ao livro de PDI e Visão Computacional — versão Python / Português.

Organização

- **Parte I — Fundamentos de PDI** — Representação, histogramas, filtragem, morfologia
- **Parte II — Visão Computacional** — Segmentação, descritores, detecção, deep learning

---

Prefácio

Este livro constitui uma **obra em constante atualização** sobre **Processamento Digital de Imagens (PDI) e Visão Computacional (VC)**, concebido como material didático interativo para cursos de graduação e pós-graduação em Computação, Engenharias e áreas correlatas.

> **!** Destaque
>
> A obra segue a metodologia descrita em Zampirolli et al. (2025) — uma extensão de Zampirolli et al. (2024), trabalho premiado na trilha de "Recursos e Ambientes Educacionais" na **EduComp 2024**. O conteúdo integra a biblioteca `morph.py`, desenvolvida pelo autor.

Um livro vivo

O material não possui caráter estático. O conteúdo passa por evolução contínua: exemplos são refinados, novas seções são incorporadas e as abordagens pedagógicas são atualizadas conforme o retorno de usuários. Por essa razão, o material é tratado como um *projeto inicial* — uma base em expansão.

Uso de ferramentas de Inteligência Artificial

A produção deste livro conta com o suporte de **ferramentas de Inteligência Artificial** (IA) para elevar a qualidade do texto e do código. A versão inicial dos capítulos foi submetida a processos de revisão por meio de assistentes, em versões gratuitas, como **DeepSeek**, **Gemini** e **Claude**. Mediante *prompts* iterativos, tais ferramentas auxiliaram no seguinte:

- Refinamento da clareza e precisão das explicações;
- Detecção e correção de erros de sintaxe, lógica e digitação em exemplos de código;
- Sugestão de melhorias na organização e fluidez textual.

Adicionalmente, **ilustrações foram geradas pelo Gemini**, com o propósito de auxiliar a visualização de conceitos abstratos da área.

Como este livro é produzido

O conteúdo é integralmente redigido em **Quarto** (armazenado na pasta `all`), um sistema de publicação científica que permite a renderização do código-fonte para múltiplos formatos:

Formato	Descrição
HTML	Versão web com navegação interativa: fzampirolli.github.io/pdi-vc
PDF	Versão para impressão ou leitura *offline*: livro.pt.py.pdf
Notebooks (`.ipynb`)	Versão processada por filtro personalizado que resolve citações e numeração de elementos, formatando referências no padrão ABNT. Compatível com **Jupyter** e **Google Colab**.

No início de cada capítulo, botões "Open in Colab" direcionam para dois ambientes:

1. **Parte Teórica:** Localizada na abertura do capítulo.
2. **Parte Prática:** Localizada na seção 💻 **Parte Prática com Exercícios de Programação** ao final do capítulo.

O *script* de pós-processamento (`gerar_notebooks_alunos.py`) prepara os arquivos para distribuição, assegurando a funcionalidade em ambientes interativos sem a necessidade de instalação local do Quarto.

A biblioteca `morph.py`

A biblioteca **`morph.py`** (Zampirolli et al., 2025) é empregada em toda a obra. Desenvolvida para fins pedagógicos, a ferramenta oferece uma interface simplificada sobre bibliotecas consolidadas, como **OpenCV**, **NumPy** e **Matplotlib**, permitindo o foco nos conceitos teóricos em detrimento de detalhes específicos de API.

> **Herança Técnica**
>
> A estrutura da `morph.py` fundamenta-se em ferramentas de morfologia matemática desenvolvidas no Brasil, como **MMachLib** (Lotufo et al., 1997) e **MMach** (Barrera et al., 1998), além da **mmorph**, empregada na obra de Dougherty; Lotufo (2003). Tais ferramentas serviram como referência para o ensino da área por décadas.

A biblioteca permite realizar leitura, exibição, conversões de cores, filtragem e morfologia matemática de forma concisa:

```
import morph as mm

img = mm.read('lena.jpg')     # leitura da imagem
mm.show(img)                  # exibição
neg = mm.neg(img)             # inversão (negativo)
mm.show(neg)
```

O código-fonte está disponível em: **github.com/fzampirolli/pdi-vc/tree/master/morph**

Exercícios de Programação (EPs) e validação automática

Cada unidade inclui **Exercícios de Programação (EPs)**. Trata-se de problemas que demandam a implementação de programas, que podem utilizar a `morph.py` ou outras bibliotecas padrão. Os EPs visam consolidar a teoria por meio de uma progressão de complexidade técnica.

Estrutura e Validação

A execução da validação ocorre por meio da classe `TestSuite`:

```
TestSuite("EP01_01.py").run()
```

O sistema compara a saída da implementação com casos de teste definidos (arquivos `.cases`) e gera um relatório de conformidade. O suporte abrange múltiplas linguagens (atualmente Python, Java, C++, C, JavaScript e R), o que possibilita a aplicação do material em diferentes contextos acadêmicos. Os mesmos EPs, com os respectivos casos de teste, estão disponíveis para a plataforma Moodle para fins de avaliação.

Idiomas e Conversão de Código

O **núcleo de referência** é escrito em **Português** com exemplos em **Python**. Versões em outros idiomas e linguagens de programação são geradas via **APIs de IA**, passando por um fluxo que automatiza a tradução

textual e a conversão de sintaxe de código.

Nota

As versões traduzidas via IA servem como **ponto de partida** e podem apresentar imprecisões, sendo recomendada a revisão antes da aplicação em sala de aula.

Código aberto

O projeto é regido por princípios de código aberto. O repositório público contém o texto, códigos, imagens e scripts: **github.com/fzampirolli/pdi-vc**

Antes de começar: Notebooks em Python

O conceito de ***Literate Programming*** (Programação Literária), proposto por Donald Knuth (Knuth, 1984), fundamenta a estrutura deste material. A lógica inverte o paradigma tradicional: o programa é escrito para a leitura humana, assemelhando-se a um ensaio, enquanto o código é extraído para execução computacional.

O conteúdo é estruturado em *Notebooks*, documentos que intercalam **células de texto** (em Markdown) e **células de código** (em Python).

- **Execução:** Células de código são identificadas por `[ ]`. A execução ocorre via Shift + Enter ou pelo botão ▷.
- **Ambientes:** O acesso pode ser feito localmente (Jupyter) ou via nuvem (Google Colab).

Nota sobre o formato

Em ambientes interativos, o código pode ser modificado e executado. Nas versões estáticas (HTML ou PDF), os blocos de código servem para fins de leitura e referência, mantendo-se a integridade das explicações.

Parte I

Parte I — Fundamentos de PDI

Capítulo 1

Fundamentos e Primeiros Passos

Este capítulo inaugura a **Parte 1** do livro, dedicada aos fundamentos do Processamento Digital de Imagens (PDI). São apresentados a representação matemática de imagens digitais e os principais métodos para sua manipulação, utilizando a linguagem Python e a biblioteca `morph.py` (Zampirolli et al., 2025).

1.1 Objetivos

Ao final deste capítulo, você será capaz de:

- Compreender a natureza física e matemática da imagem digital $f(x, y)$.
- Identificar as faixas do espectro eletromagnético relevantes para PDI.
- Configurar o ambiente de desenvolvimento em Python.
- Realizar operações básicas: leitura, exibição e salvamento de imagens.
- Manipular estruturas de matrizes (NumPy) sem cair em armadilhas de memória.
- Acessar e modificar intensidades de pixels individualmente.
- Aplicar limiarização manual.

1.2 Antes de começar: *Notebooks* em Python

Este material foi construído sob o conceito de ***Literate Programming*** (Programação Literária), idealizado por Donald Knuth na década de 1980 (Knuth, 1984). Knuth — também criador do sistema **TeX** para tipografia digital — propôs que os programas fossem escritos como uma narrativa lógica para seres humanos, intercalando código e documentação.

Para executar uma célula, pressione Shift + Enter ou clique no botão ▷.

> ℹ Nota sobre o formato
>
> Nas versões renderizadas (PDF ou HTML), o código é apresentado em blocos estáticos para fins de leitura e referência. A execução interativa requer o acesso via Google Colab (disponível no topo da página) ou em ambiente local via VSCode ou Jupyter Notebook.

1.3 Fundamentos

O estudo de sistemas baseados em imagens compreende um ecossistema de disciplinas integradas que transformam dados visuais brutos em conhecimento estruturado. Enquanto algumas áreas focam na geração de representações, outras dedicam-se ao tratamento e à análise desses dados para dar suporte a aplicações tecnológicas complexas.

O diagrama apresentado no Figure 1.1 estabelece a distinção e complementaridade entre o Processamento de Imagens (PDI) e a Visão Computacional (VC). O PDI, destacado em **verde**, tem como foco a transformação

de imagem para imagem, visando melhoria de qualidade ou pré-processamento, como a remoção de ruídos e realce de contraste.

Em contrapartida, a VC, assinalada em **azul**, foca na interpretação do conteúdo visual para extrair modelos ou informações, como o reconhecimento de objetos e gestos. A região de intersecção ilustra a sinergia entre as áreas, onde o PDI prepara os dados visuais para a interpretação pela VC. O mapa também demonstra as interconexões de ambas as disciplinas com áreas como Robótica, Computação Gráfica, Inteligência Artificial e Neurociência.

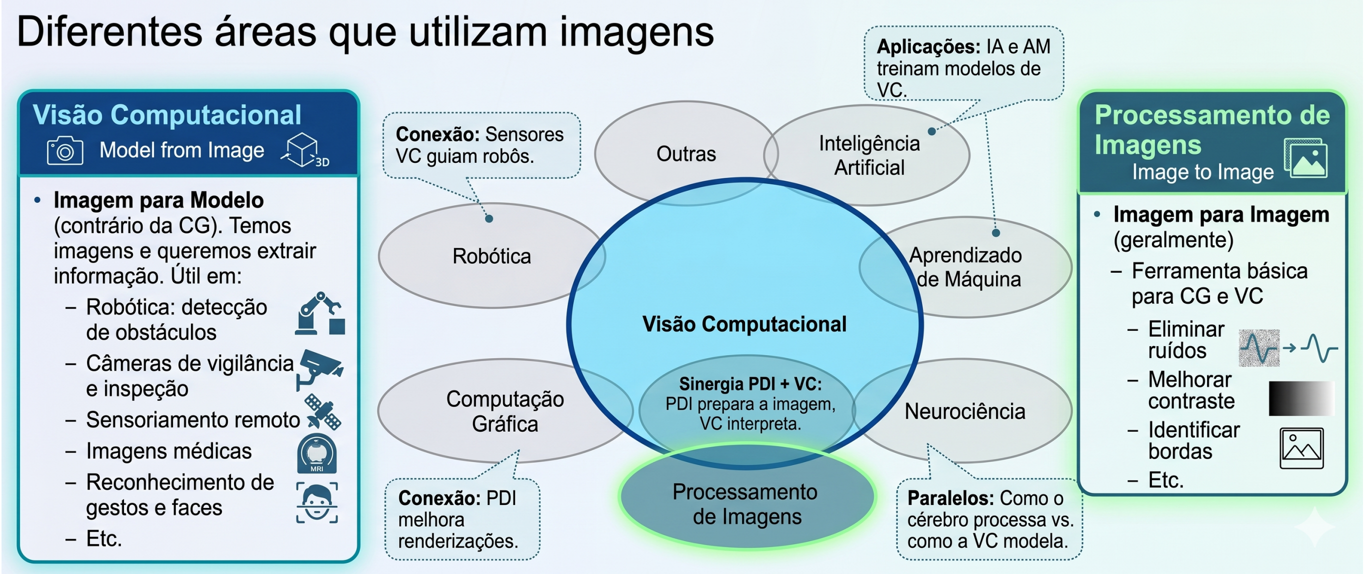


Figura 1.1: Diagrama relacional detalhando as distinções fundamentais, sinergias e interconexões entre Processamento de Imagens e Visão Computacional no contexto de sistemas baseados em imagens.

1.3.1 👁 Visão Computacional

- **Foco:** *Imagem* → *Modelo* (caminho inverso da Computação Gráfica).
- **Objetivo:** Extrair informação de alto nível a partir de imagens ou vídeos.
- **Aplicações típicas:**
  - Robótica - detecção de obstáculos, localização e navegação autônoma.
  - Vigilância e inspeção - reconhecimento de eventos, leitura de placas, controle de qualidade.
  - Sensoriamento remoto - análise de imagens de satélite, mapeamento ambiental.
  - Imagens médicas - detecção de tumores, segmentação de órgãos, auxílio ao diagnóstico.
  - Interação humano-computador - reconhecimento de gestos, expressões faciais, rastreamento ocular.
- **Relação com outras áreas:** utiliza técnicas de Aprendizado de Máquina e IA para classificar e interpretar cenas; serve como "olhos" da Robótica.

1.3.2 🖼 Processamento Digital de Imagens (PDI)

- **Foco:** *Imagem* → *Imagem* (geralmente - transformação de uma imagem em outra).
- **Objetivo:** Melhorar a qualidade visual ou extrair características de baixo nível.
- **Aplicações comuns:**
  - Eliminação de ruídos (filtros de média, mediana, gaussiano).
  - Melhoria de contraste (equalização de histograma, ajuste gamma).
  - Detecção de bordas (Sobel, Canny, Laplaciano).
  - Segmentação (limiarização, crescimento de regiões, *watershed*).

  - Transformações geométricas (redimensionamento, rotação, correção de perspectiva).
- **Relação com outras áreas:**
  - É a **base** para a maioria dos sistemas de Visão Computacional (pré-processamento).
  - A Computação Gráfica frequentemente aplica PDI para pós-processamento (ex.: suavização, realce).
  - Técnicas de IA podem otimizar parâmetros de processamento (ex.: aprendizado de filtros).

1.3.3 Como as outras áreas se conectam

Área	Relação com PDI e VC
Inteligência Artificial	Fornece modelos (redes neurais, SVM) que interpretam saídas da VC.
Robótica	Consome dados de VC para tomar decisões (navegação, manipulação).
Aprendizado de Máquina	Usa descritores extraídos pelo PDI/VC para treinar classificadores.
Computação Gráfica	Caminho inverso: *modelo* → *imagem*; muitas vezes aplica PDI para renderização realista.
Neurociência	Inspira modelos de PDI (ex.: filtros semelhantes a células ganglionares da retina).

1.4 Etapas do PDI

As etapas do PDI são apresentadas na Figure 1.2, que podem ser compreendidas como uma cadeia de transformações que reduz a redundância dos dados em busca de significado:

- **Baixo Nível:** Atua diretamente sobre os pixels da imagem ruidosa para realizar melhorias e filtragens, gerando como saída uma imagem limpa ou realçada.
- **Médio Nível:** Recebe a imagem tratada e realiza a segmentação e descrição, transformando a matriz de pixels em atributos estruturados (forma, tamanho e textura).
- **Alto Nível:** Utiliza a tabela de atributos para alimentar processos de lógica e inteligência artificial, resultando na decisão ou reconhecimento final (como o diagnóstico médico).

A Figure 1.3 detalha a sequência completa do processamento, desde a captura até a interpretação. O fluxo inicia-se na Aquisição da Imagem (1) e prossegue com o Melhoramento (2) e a Restauração (3). Em seguida, o conteúdo é isolado pela Segmentação (4) e refinado pela Morfologia (5). A transição crucial ocorre na Representação e Descrição (6), onde objetos visuais são convertidos em dados matemáticos (área, perímetro etc.), permitindo o Reconhecimento (7). Processos auxiliares incluem o Processamento de Imagem Colorida e a Compressão, que contribuem para a eficiência do armazenamento e da análise.

1.5 Formação da Imagem e o Espectro

O processo de formação de uma imagem é fundamentado na interação entre a matéria e a energia radiante. Essencialmente, uma imagem é concebida quando um **sensor registra a radiação** resultante da interação com um objeto físico. No contexto da visão humana e da fotografia convencional, este fenômeno depende de uma fonte de luz que ilumine a cena; as características dos objetos são então codificadas através das **variações de intensidade e cor** da luz que atinge o sensor, conforme ilustrado na Figure 1.4.

A **luz visível** ocupa apenas uma pequena faixa do espectro eletromagnético — entre **380 nm (violeta)** e **750 nm (vermelho)** — conforme ilustrado na Figure 1.5. Sensores digitais convencionais operam nessa mesma janela, mas equipamentos especializados podem captar radiações invisíveis ao olho humano, como o infravermelho e os raios X. Em PDI, a imagem formada depende diretamente da sensibilidade espectral do sensor utilizado.

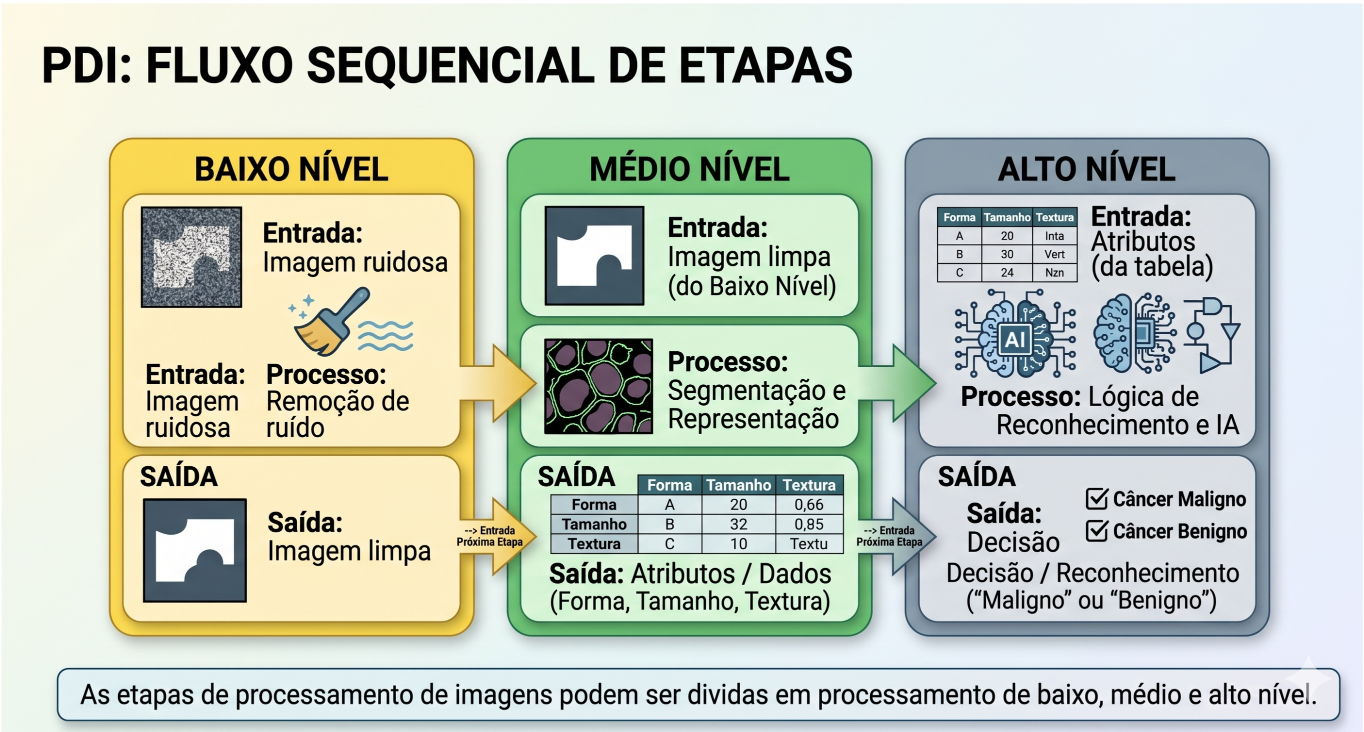


Figura 1.2: Representação do fluxo sequencial de processamento: a saída de cada nível torna-se a entrada do nível subsequente.

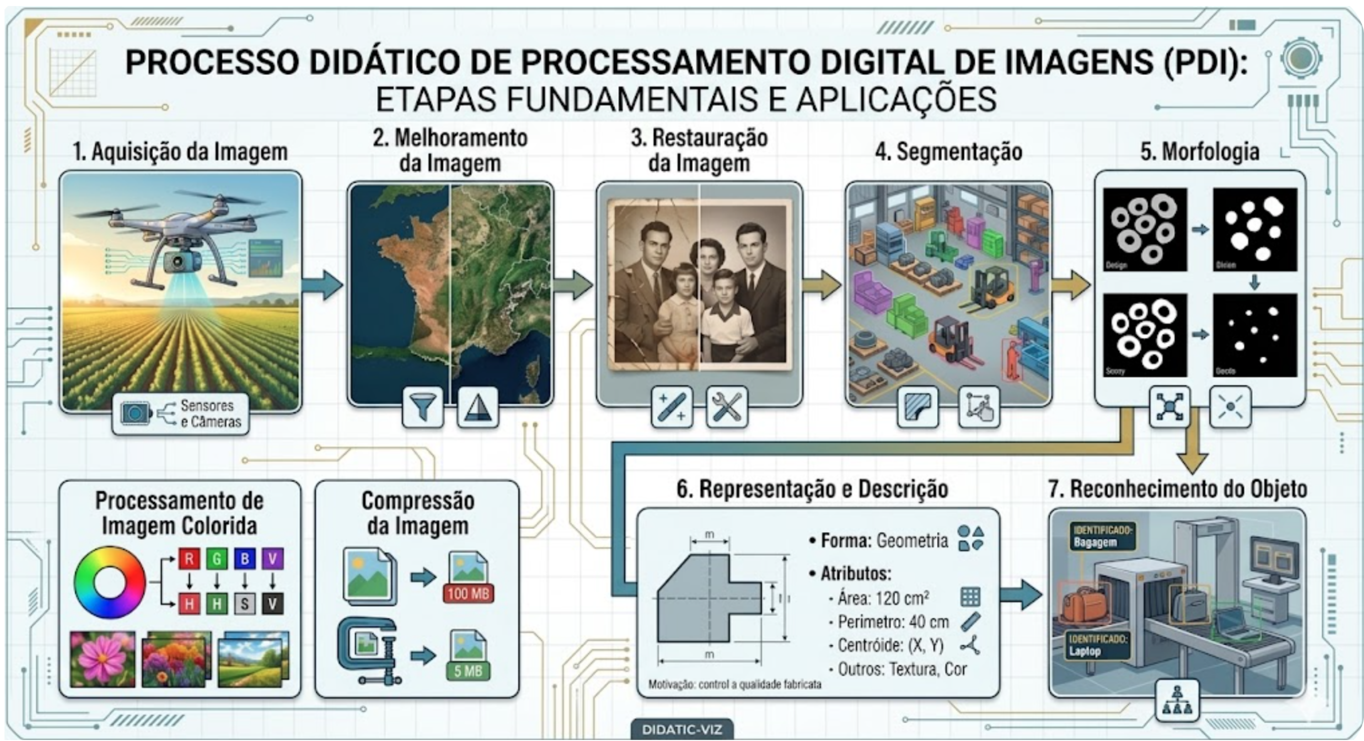


Figura 1.3: Fluxo detalhado do PDI: da aquisição sensorial à extração de atributos e reconhecimento automatizado, incluindo em processamento colorido e compressão.

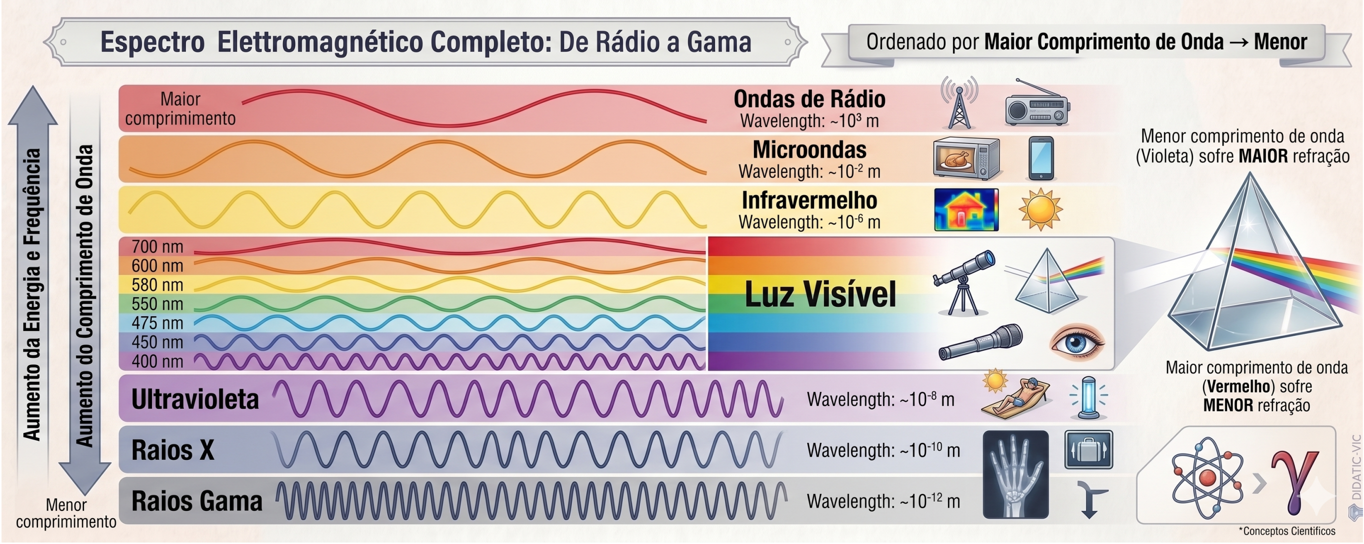


Figura 1.4: Representação do espectro visível e sua posição em relação às demais radiações eletromagnéticas, destacando a variação dos comprimentos de onda de 380 nm a 750 nm.

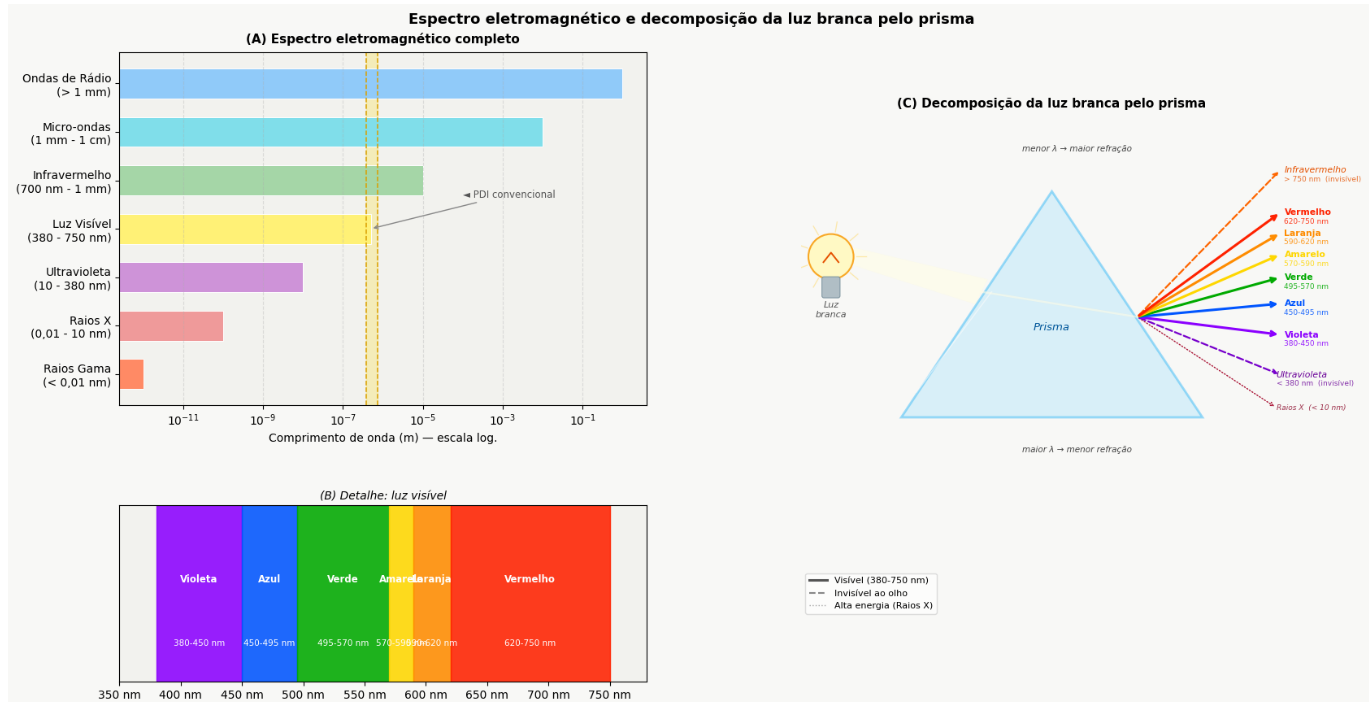


Figura 1.5: (A) Espectro eletromagnético completo em escala logarítmica, com destaque para a faixa visível. (B) Detalhe da luz visível (380-750 nm) e suas cores componentes. (C) Decomposição da luz branca pelo prisma: menor comprimento de onda sofre maior refração, separando UV, visível e infravermelho.

1.6 O que é uma Imagem Digital?

Uma **imagem digital** é formada por uma grade de **pixels** (*Picture Elements*), onde cada pixel é a menor unidade elementar da imagem.

O que é um Pixel?

Um **pixel** é a menor unidade endereçável que compõe uma imagem digital. Cada pixel ocupa uma posição única na grade e armazena um ou mais valores numéricos que representam sua intensidade ou cor.

Representação Matemática

Diferente de uma função contínua, o domínio de uma imagem digital é um plano retangular finito $\mathbb{E} \subset \mathbb{Z}^2$, que representa a grade de amostragem. Este domínio é indexado por coordenadas inteiras:

$$\mathbb{E} = \{(x, y) \in \mathbb{Z}^2 \mid 0 \leq x < L,\ 0 \leq y < H\} \tag{1.1}$$

Onde:

- L: representa a largura da imagem (número de colunas).
- H: representa a altura da imagem (número de linhas).

A imagem digital é uma função que associa cada par de coordenadas (x, y) a um ou mais valores que descrevem a aparência do pixel.

$$f : \mathbb{E} \rightarrow \mathcal{V} \tag{1.2}$$

O conjunto $\mathcal{V}$ define os valores possíveis para o pixel (codomínio), variando conforme o tipo de imagem, como demonstrado na Table 1.2.

Tabela 1.2: Principais tipos de imagem digital e seus respectivos conjuntos de valores possíveis para cada pixel.

Tipo de imagem	$\mathcal{V}$ (valores do pixel)	Representação
Binária	$\{0, 1\}$ ou $\{0, 255\}$	■□
Tons de cinza	$\{0, 1, \ldots, 255\}$	[ ] [=] [#] ■
Colorida (RGB)	$\{0, \ldots, 255\}^3$ (triplas ordenadas de valores)	■■■

Exemplo prático: Uma imagem colorida (RGB) é representada matematicamente por uma função que retorna três valores para cada pixel. No computador, isso resulta em três matrizes sobrepostas (canais R, G e B), onde cada célula contém um valor de intensidade para aquele canal específico no pixel.

1.7 Configuração do Ambiente

Este material fundamenta-se no ecossistema científico do Python, com destaque para o **NumPy** (matemática matricial), **OpenCV** (visão computacional padrão de mercado) e a biblioteca **morph.py** (Zampirolli et al., 2025), desenvolvida especificamente para fins didáticos neste livro.

Atualmente, o projeto disponibiliza a versão em **Python** e **Português**. A arquitetura do sistema, no entanto, foi projetada para expansão futura, permitindo a adaptação automatizada para outras linguagens de programação (como C++ e Java) e idiomas, por meio de APIs de tradução.

Os **Exercícios de Programação (EPs)** do final de cada capítulo integram o módulo `testsuite.py`, já disponível para práticas em Python, C, C++, Java, JavaScript e R, garantindo a validação imediata das soluções propostas nos notebooks de estudo.

1.7.1 📦 Bibliotecas Utilizadas

Biblioteca	Função principal
`numpy`	Representação matricial de imagens
`opencv-python`	Leitura, escrita e operações de visão computacional
`matplotlib`	Visualização de imagens e gráficos
`morph.py`	Abstração didática das operações de PDI

ℹ Versões do `morph.py`

A biblioteca `morph.py` possui duas versões públicas:

- **Versão 1.0** — versão original, publicada junto ao artigo apresentado no EduComp 2024: https://github.com/fzampirolli/morph
- **Versão 1.1** — versão compacta, utilizada nas atividades deste livro: https://github.com/fzampirolli/pdi-vc/blob/master/morph/morph.py

A versão 1.1 foi necessária para uso nas atividades do **Moodle/VPL** (Laboratório Virtual de Programação). Na versão 1.0, bibliotecas como `matplotlib`, `requests` e `skimage` eram carregadas no momento do `import morph`, causando erro de memória (`Jail: out of memory, 128MiB`) no ambiente restrito do VPL.

Na versão 1.1, esses imports foram convertidos para **carregamento lazy**: cada biblioteca é importada apenas dentro do método que a utiliza, e somente quando esse método é de fato chamado. Assim, um simples `import morph` carrega apenas `numpy` e `cv2`, que são suficientes para a maioria dos EPs.

```python
import os, importlib, urllib.request, numpy as np, matplotlib.pyplot as plt

# URLs do repositório
BASE_URL = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph"
for f in ["morph.py"]:
    if not os.path.exists(f):
        urllib.request.urlretrieve(f"{BASE_URL}/{f}", f)

import morph
importlib.reload(morph)
from morph import mm

# Correção: Verifica se o atributo existe antes de imprimir,
# ou simplesmente confirma o carregamento do módulo.
version = getattr(morph, "__version__", "local_file")
print(f" Ambiente pronto. Módulo 'morph' carregado com sucesso.")
print(f" Funções disponíveis no mm: \n{dir(mm)[-5:]}...")
```

```
 Ambiente pronto. Módulo 'morph' carregado com sucesso.
 Funções disponíveis no mm:
['verifyBoundBox', 'watershed', 'watershed0', 'watershedB', 'write']...
```

1.8 Fundamentos de Matrizes - atenção à cópia de referências

Como uma imagem digital é uma matriz, precisamos saber criá-las corretamente. Em Python, existe uma armadilha comum ao usar o operador * em listas:

⚠ Atenção à cópia de referências

Ao fazer `m = [[0]*2]*3`, você não cria 3 linhas independentes, mas sim 3 referências para a **mesma** linha. Alterar um valor em uma linha alterará todas as outras!

Para visualizar esse comportamento, você pode testar o código no Python Tutor e comparar com a forma correta de criar matriz com listas: `m = [[0]*2 for _ in range(3)]`.

A forma recomendada e padrão em Processamento Digital de Imagens é usar **NumPy**, que cria matrizes com dados independentes e oferece eficiência computacional. O código a seguir mostra diferentes formas de criação de imagens sintéticas — o resultado é exibido na Figure 1.6.

```
# Criando uma imagem preta (zeros) de 5x5 pixels
img_preta = np.zeros((5, 5), dtype='uint8')

# Criando uma imagem branca (255) de 5x5 pixels
img_branca = np.ones((5, 5), dtype='uint8') * 255

# Criando uma imagem aleatória para testes (Ruído)
img_random = mm.randomImage(4, 6, maxValue=255)

print("Matriz Aleatória Gerada:")
print(mm.drawImage(img_random))

mm.show(img_random, title="Exemplo: Captura RGB")  # menor
```

```
Matriz Aleatória Gerada:
226 156  71 174 248  40
121 173  36 183  57 156
 44  65 200 214 178  93
250 139  31 165 105 155
```

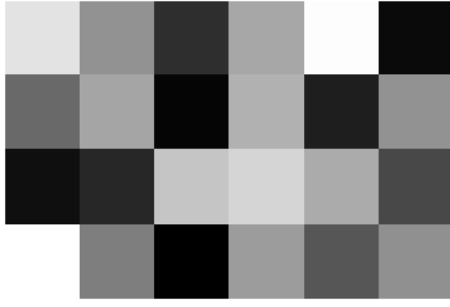


Figura 1.6: Exemplo de imagem sintética gerada por matriz aleatória 4x6.

1.9 Lendo e Exibindo Imagens

Em Python, as imagens são lidas como *arrays* do NumPy. Isso significa que toda a potência da álgebra linear está disponível para o processamento de imagens.

A operação mais básica em PDI é a leitura de uma imagem. A função `mm.read()` da `morph.py` aceita caminhos locais e URLs, ver o resultado na Figure 2.6.

```
import os
import numpy as np

url     = "https://upload.wikimedia.org/wikipedia/commons/8/87/Mandrillus_sphinx_339428057.jpg"
```

```
caminho = "imagens/mandrill.jpg"

if not os.path.exists(caminho):
    os.makedirs("imagens", exist_ok=True)
    img_obj = mm.read(url, pil=True)
    mm.write(img_obj, caminho)
else:
    img_obj = mm.read(caminho, pil=True)

img = np.array(img_obj)

print(f"Dimensões (H, W, Canais): {img.shape}")
print(f"Tipo de dado: {img.dtype}")

mm.show(img, title="Exemplo: Captura RGB")
```

```
Dimensões (H, W, Canais): (1365, 2048, 3)
Tipo de dado: uint8
```

Exemplo: Captura RGB

Figura 1.7: Mandrill (Mandrillus sphinx) in Gabon GPS: (-1.93867, 13.09674). Crédito: Julien Renoult (CC BY 4.0).

Alternativa: Baixando a imagem localmente

Se por algum motivo a leitura direta da URL não funcionar (firewall, restrições de rede, ou se você preferir trabalhar com arquivos locais), uma abordagem simples é **baixar a imagem** usando o comando `wget` (disponível em ambientes Linux, macOS e no Google Colab) e depois carregá-la com `mm.read()` a partir do arquivo salvo.

```
!wget -O mandrill.png https://upload.wikimedia.org/wikipedia/commons/8/87/Mandrillus_sphinx_339428057.jp
```

```
img = mm.read('mandrill.png')
mm.show(img, title="Exemplo: Captura RGB (arquivo local)")
```

Explicação:

- `!wget -O mandrill.png <URL>` baixa a imagem da Wikimedia e a salva com o nome `mandrill.png`.
- `mm.read('mandrill.png')` lê o arquivo local (sem necessidade de cabeçalhos HTTP ou tratamento especial).
- Essa estratégia é útil quando você quer **evitar dependências de rede** durante a execução ou **reutilizar a mesma imagem** várias vezes sem novo download.

Nota

O prefixo `!` funciona no Jupyter Notebook, JupyterLab e Google Colab. Em um terminal comum, basta remover o `!`. Se o `wget` não estiver instalado, instale-o com `apt install wget` (Linux) ou use `curl -o mandrill.png <URL>` como alternativa.

1.10 Conversão de Tipos e Limiarização

Conforme vimos, uma imagem colorida no espaço RGB é representada pela função:

$$f : \mathbb{E} \to \{0, 1, \dots, 255\}^3$$

Ou seja, para cada pixel (x, y), temos três valores (R, G, B) que definem sua cor.

Conversão para Tons de Cinza

Para converter uma imagem RGB em tons de cinza (*grayscale*), é necessário combinar os três canais em um único valor de intensidade g, que representa o brilho percebido. Como o olho humano não é igualmente sensível ao vermelho, verde e azul, utiliza-se uma **média ponderada**. O padrão ITU-R BT.601 (ITU-R, 2011) define os seguintes pesos:

$$g = 0.299\,R + 0.587\,G + 0.114\,B \tag{1.3}$$

Após o cálculo, o valor g é arredondado para o inteiro mais próximo e ajustado ao intervalo $[0, 255]$. O resultado é uma nova imagem, agora em tons de cinza, representada por:

$$f_{\text{cinza}} : \mathbb{E} \to \{0, 1, \dots, 255\}$$

Limiarização (*Thresholding*)

A partir da imagem em tons de cinza $f_{\text{cinza}}(x, y)$, uma operação fundamental é a **limiarização**, que produz uma imagem **binária** (apenas preto e branco). Para isso, escolhe-se um valor de corte T (geralmente no intervalo $[0, 255]$) e define-se:

$$f_{\text{bin}}(x, y) = \begin{cases} 255 & \text{se } f_{\text{cinza}}(x, y) > T \\ 0 & \text{caso contrário} \end{cases} \tag{1.4}$$

Exemplo: Com $T = 128$, pixels com intensidade acima de 128 tornam-se brancos (255); os demais tornam-se pretos (0).

A limiarização é amplamente usada para **segmentar objetos do fundo**, extrair bordas ou criar máscaras binárias para processamento posterior.

> **Nota:** O valor 255 representa o branco máximo em imagens de 8 bits, enquanto 0 representa o preto absoluto.

Exemplo prático de conversão e limiarização

A Figure 1.8 ilustra os principais passos para transformar uma imagem colorida em tons de cinza e, em seguida, convertê-la em uma imagem binária por limiarização. O código a seguir implementa essas etapas:

```
import matplotlib.pyplot as plt

# 1. Converter para Tons de Cinza
img_gray = mm.gray(img)

# 2. Aplicar limiar (Pixels > 128 tornam-se 255, outros 0)
limiar = 128
img_binaria = mm.threshold(img_gray, limiar)

# Uso da nova função
mm.show(
    [img, img_gray, img_binaria],
    titles=["Original", "Tons de Cinza", f"Binária (T={limiar})"],
    cols=3
)
```

Figura 1.8: Processamento básico de imagens: (a) imagem original, (b) imagem em tons de cinza, (c) imagem binarizada por limiar (T=128).

1.11 Limiarização pelo método de Otsu

Conforme apresentado na Equation 1.4, a limiarização converte uma imagem em tons de cinza para binária usando um valor de corte T. Até agora fixamos $T = 128$ manualmente.

```
img_bin_fixo = mm.threshold(img_gray, T=128)
```

Entretanto, a escolha manual de T nem sempre é trivial. A biblioteca `mm` oferece uma alternativa automática: quando o parâmetro **`limiar` não é fornecido**, a função `mm.threshold(img_gray)` calcula o valor de T pelo **método de Otsu** (Otsu, 1979). Esse método, que será detalhado em capítulos futuros, maximiza a variância entre classes do histograma, separando automaticamente os pixels de objeto e fundo.

O código abaixo compara a limiarização manual ($T = 128$) com a automática (Otsu), mostrando também o valor de T calculado:

```
import cv2
# Limiarização com T fixo (manual)
T_fixo = 128
img_bin_fixo = mm.threshold(img_gray, T_fixo)

# Limiarização pelo método de Otsu (T automático)
T_otsu, img_bin_otsu = cv2.threshold(img_gray, 0, 255,
                                      cv2.THRESH_BINARY + cv2.THRESH_OTSU)
print(f'Limiar calculado por Otsu: T = {T_otsu}')
```

```
# ou simplesmente:
# img_bin_otsu = mm.threshold(img_gray)

# Exibição lado a lado
mm.show(
    [img_gray, img_bin_fixo, img_bin_otsu],
    titles=["Tons de Cinza", f"Binária (T={T_fixo})", f"Binária (Otsu, T={T_otsu})"],
    cols=3
)
```

```
Limiar calculado por Otsu: T = 96.0
```

Figura 1.9: Comparação entre limiarização manual (T=128) e automática (Otsu) sobre a imagem em tons de cinza.

A Figure 1.9 mostra que o limiar obtido por Otsu se adapta automaticamente à imagem, resultando em uma binarização mais eficiente do que um valor fixo, especialmente quando as intensidades do objeto e do fundo são bem separadas no histograma. Essa técnica é amplamente utilizada em sistemas de visão computacional para binarização de documentos, detecção de objetos e pré-processamento de imagens.

Simplicidade da biblioteca `morph.py`

Enquanto o OpenCV exige a chamada completa:

```
T_otsu, img_bin = cv2.threshold(img_gray, 0, 255,
                                cv2.THRESH_BINARY + cv2.THRESH_OTSU)
```

a biblioteca `mm` abstrai toda essa complexidade: basta chamar `mm.threshold(img_gray)`. O limiar de Otsu é calculado automaticamente e a imagem binária é retornada diretamente. Essa abordagem permite concentrar-se no conceito, não nos detalhes de implementação.

1.12 Acesso a Pixels

Em Python com NumPy, uma imagem é representada como um **array multidimensional**. Para acessar um pixel específico, utilizam-se as coordenadas **linha** (eixo Y) e **coluna** (eixo X): `img[linha, coluna]`.

O código abaixo demonstra como obter o valor de um pixel em uma imagem RGB e em sua versão em tons de cinza, além de criar uma pequena imagem sintética para visualizar a estrutura de uma matriz de pixels.

```
import numpy as np
import matplotlib.pyplot as plt

# Coordenadas do pixel que queremos examinar
r, c = 100, 100

# Acessa o pixel na imagem em tons de cinza (um valor escalar)
```

```
pixel_cinza = img_gray[r, c]

# Acessa o pixel na imagem colorida RGB (vetor de 3 valores)
pixel_rgb = img[r, c]

print(f'Pixel na posição ({r},{c}):')
print(f'  - Tons de cinza : {pixel_cinza}')
print(f'  - RGB           : R={pixel_rgb[0]}, G={pixel_rgb[1]}, B={pixel_rgb[2]}')

# Cria uma imagem sintética 5×5 com todos os pixels pretos (0)
syn = np.zeros((5, 5), dtype=np.uint8)
# Torna o pixel central (linha 2, coluna 2) branco (255)
syn[2, 2] = 255

print("\nMatriz da imagem sintética 5×5:")
print(syn)

# Exibe a imagem sintética
# plt.figure(figsize=(3, 3))
# plt.imshow(syn, cmap='gray', vmin=0, vmax=255)
# plt.title("Imagem sintética 5×5\n(pixel central branco)")
# plt.axis('off')
# plt.tight_layout()
# plt.show()
# ou simplesmente:

mm.show(syn, title="Exemplo: Imagem Sintética 5x5")
```

```
Pixel na posição (100,100):
  - Tons de cinza : 64
  - RGB           : R=77, G=66, B=20

Matriz da imagem sintética 5×5:
[[  0   0   0   0   0]
 [  0   0   0   0   0]
 [  0   0 255   0   0]
 [  0   0   0   0   0]
 [  0   0   0   0   0]]
```

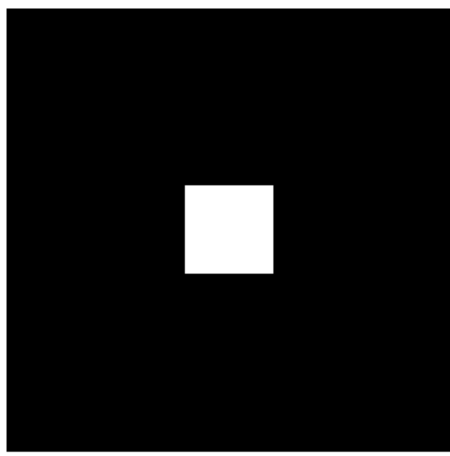


Figura 1.10: Exemplo de matriz sintética 5×5 com pixel central branco.

Explicação linha a linha:

1. `img_gray[r, c]` - retorna um único número inteiro entre 0 e 255, correspondente ao nível de cinza naquela posição.

2. `img[r, c]` - retorna uma tupla ou array com três valores (R, G, B).

3. `np.zeros((5,5), dtype=np.uint8)` - cria uma matriz 5×5 preenchida com zeros (preto).

4. `syn[2, 2] = 255` - altera o elemento central da matriz para 255 (branco), demonstrando como modificar um pixel.

5. A exibição com `plt.imshow` revela um pequeno quadrado com um ponto branco no meio, ilustrando visualmente a estrutura discreta da imagem.

A Figure 1.10 mostra os valores impressos e a imagem sintética. A indexação em Python é *zero-based*; portanto, o canto superior esquerdo corresponde a `(0,0)`. A ordem **linha** × **coluna** corresponde à estrutura matricial da imagem: primeira dimensão → altura (Y), segunda dimensão → largura (X).

1.13 Resumo

Neste capítulo, apresentaram-se os fundamentos de representação de imagens digitais: a definição de pixel, a estruturação de imagens em matrizes e o impacto da amostragem e da quantização na qualidade final:

- **Imagem digital** = função $f(x, y)$ que mapeia coordenadas para intensidades (escalares ou vetoriais).
- **Domínio**: conjunto finito $\mathbb{E} = \{(x, y) \in \mathbb{Z}^2 \mid 0 \leq x < L,\ 0 \leq y < H\}$.
- **Tipos principais**: binária ($\mathcal{V} = \{0, 255\}$), tons de cinza ($\mathcal{V} = [0, 255]$) e RGB ($\mathcal{V} = [0, 255]^3$).
- A biblioteca `morph.py` (ou `mm`) oferece funções didáticas para operações básicas de PDI, como `mm.gray()`, `mm.threshold()`, `mm.show_multiple()`.
- **Armadilha NumPy**: nunca use `[[0]*n]*m` para criar matrizes — use sempre `np.zeros()` ou `np.ones()`.
- **Limiarização** converte tons de cinza em binária; o **método de Otsu** determina o valor de corte automaticamente maximizando a variância entre classes.
- Acesso a pixels via `img[linha, coluna]`, com indexação **zero-based**.

O Capítulo 2 abordará **histogramas e equalização de contraste**.

1.14 🤖 Uso do NotebookLM como Tutor Complementar

Nesta edição, além dos notebooks interativos no Google Colab, incentivamos o uso do **NotebookLM** como ferramenta complementar de aprendizagem. Essa ferramenta de IA utiliza exclusivamente os documentos fornecidos pelo autor como base de conhecimento, garantindo respostas coerentes com o conteúdo do livro.

Para cada capítulo, preparamos um projeto específico na plataforma. Para uma experiência de estudo ampliada, utilize o acesso abaixo:

> 🎓 Estude com o Tutor Inteligente
>
> Para interagir com o conteúdo deste capítulo, acesse o link a seguir. O ambiente contém materiais didáticos em diferentes formatos, gerados a partir do **PDF** do capítulo. Na plataforma, explore especialmente as opções **Guia de Estudo** e **Conversa** para aprofundar sua compreensão.
> 🚀 ACESSAR NOTEBOOKLM: CAPÍTULO 01

Funcionalidades Disponíveis na Plataforma

O **NotebookLM** oferece uma suíte avançada de ferramentas baseadas em IA para transformar o conteúdo estático do livro em uma experiência de aprendizado dinâmica e multimídia. Conforme ilustrado na

Figure 1.11, a plataforma utiliza técnicas de **RAG** (*Retrieval-Augmented Generation*), fundamentadas no trabalho de Lewis et al. (2020), para basear as respostas estritamente nos documentos fornecidos e minimizar a ocorrência de alucinações.

As principais funcionalidades incluem:

- **Resumos Multimodais (Áudio e Vídeo)**: Geração de conversas naturais entre especialistas no formato de **Resumo em Áudio** (estilo *podcast*) e **Resumo em Vídeo**, discutindo os temas centrais do capítulo, como as diferenças entre processamento de imagens e visão computacional, ou a interpretação de transformações como a limiarização e o método de Otsu.
- **Visualização de Estruturas (Mapa Mental e Infográfico)**: Criação automática de diagramas que conectam visualmente os conceitos, por exemplo, o fluxo de processamento desde a captura da imagem digital, passando pela conversão para tons de cinza, limiarização e segmentação binária.
- **Ferramentas de Avaliação (Teste e Cartões Didáticos)**: Geração de **Testes** de múltipla escolha e **Cartões Didáticos** (*flashcards*) para fixação de conhecimento, baseados no texto autoral (ex.: perguntas sobre a fórmula de conversão RGB→cinza ou sobre o funcionamento do limiar global e de Otsu).
- **Apoio à Apresentação (*Slides* e Relatórios)**: Auxílio na estruturação de **Apresentações de *Slides*** e na redação de **Relatórios** técnicos e ***Briefings***, facilitando a comunicação de resultados de experimentos com imagens.
- **Análise de Dados (Tabela de Dados)**: Organização de dados extraídos do texto em tabelas estruturadas, auxiliando na compreensão de exemplos práticos, como a comparação entre diferentes valores de limiar.
- ***Chat* Contextualizado**: Permite o questionamento direto sobre o código e a teoria, como: *"Como implementar a conversão de RGB para tons de cinza usando os pesos do padrão ITU-R BT.601?"* ou *"O que acontece com a imagem binária se eu escolher um limiar T=200 em vez de T=128?"*.

Figura 1.11: Estúdio do NotebookLM.

1.15 Lista de Exercícios

1. (15%) Com suas próprias palavras, defina **imagem digital** e **pixel**. Dê um exemplo concreto de como uma imagem colorida (RGB) é representada matricialmente no computador.

2. (15%) Explique as diferenças entre **imagem binária**, **tons de cinza (8 bits)** e **colorida RGB**, indicando a faixa de valores possíveis para cada pixel em cada tipo.

3. (20%) Considerando a fórmula de conversão RGB → tons de cinza do padrão ITU-R BT.601:

$$g = 0.299\,R + 0.587\,G + 0.114\,B$$

Calcule o valor do pixel em tons de cinza para $(R, G, B) = (80, 180, 30)$. Arredonde para o inteiro mais próximo.

4. (20%) O que é **limiarização** (*thresholding*)? Explique a diferença entre escolher um limiar T fixo (ex.: $T = 128$) e utilizar o **método de Otsu** para determinação automática do limiar. Em poucas palavras, como o método de Otsu escolhe o limiar?
5. (15%) No contexto da biblioteca didática `mm` discutida no capítulo, responda:
   - a) (7,5%) Como se acessa o valor do pixel na posição (linha=50, coluna=60) de uma imagem em tons de cinza `img_gray`?
   - b) (7,5%) Qual é a vantagem de usar `mm.threshold(img_gray)` sem passar o limiar? Compare com a chamada equivalente no OpenCV.
6. (15%) O que a propriedade `img.shape` retorna para uma imagem NumPy no formato RGB? Dê um exemplo concreto com uma imagem de 640×480 pixels.

Referências do Capítulo

A fundamentação teórica deste capítulo compreende as seguintes obras de processamento de imagens e visão computacional:

- Gonzalez; Woods (2018) para os fundamentos de **Processamento Digital de Imagens** (PDI).
- Singh (2019) para a implementação prática de métodos de **processamento e análise de imagens**.
- Szeliski (2022) para o estudo de **visão computacional** e algoritmos fundamentais.
- Bradski; Kaehler (2008) para a aplicação da biblioteca **OpenCV** em ambiente Python.
- Lewis et al. (2020) para o conceito de **geração aumentada por recuperação (RAG)**, utilizado no suporte ao processamento de informações deste material.

1.16 💻 Parte Prática com Exercícios de Programação

🎯 Objetivo deste Caderno

Os **Exercícios de Programação (EPs)** apresentados a seguir podem ser submetidos também em atividades do Moodle (atividades VPL) que fornecem *feedback* automático.

Este caderno foi desenvolvido para superar limitações de uso do Moodle. Com ele, deve-se:

1. **Desenvolver:** Escrever e editar sua solução diretamente no ambiente Colab.
2. **Validar:** Testar seu código localmente utilizando os **mesmos casos de teste** do Moodle.
3. **Organizar:** Salvar seus códigos das atividades VPL de forma segura.
4. **Avaliar:** Quando estiver conectado ao Moodle, basta copiar sua solução e clicar em **Avaliar** no Moodle para registrar sua nota oficial.

---

§ Instruções Passo a Passo

Em um ambiente de execução (como VSCode, Jupyter ou Colab), siga a ordem abaixo para configurar o ambiente e validar seus exercícios:

Preparação do Ambiente

Execute a célula de código abaixo para baixar `morph.py` e `testsuite.py` do repositório da disciplina — apenas se ainda não existirem no diretório local. Com ambos em `./`, o notebook e os subprocessos do

`TestSuite` encontram o módulo sem configurações extras de caminho.

Nota: O script `testsuite.py` buscará automaticamente os casos de teste em `all/{cap}/cases` no GitHub.

Escrevendo o Código

Salve sua solução em uma célula de código usando o comando mágico `%%writefile`. O nome do arquivo deve seguir o padrão `EPX_Y.*`, onde `X` é o capítulo, `Y` é o exercício e `*` é a extensão da linguagem.

Exemplo: `%%writefile EP01_01.py`

Download

Baixe `morph.py` e `testsuite.py` executando a célula abaixo:

```
import os, sys, importlib, inspect, urllib.request

# URLs do repositório
BASE_URL = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph"
for f in ["morph.py", "testsuite.py"]:
    if not os.path.exists(f):
        urllib.request.urlretrieve(f"{BASE_URL}/{f}", f)

import morph, testsuite
importlib.reload(morph); importlib.reload(testsuite)
from morph import mm
from testsuite import TestSuite

print(f" Ambiente pronto. Morph: {morph.__version__} | TestSuite: {testsuite.__version__}")
```

```
 Ambiente pronto. Morph: 1.1.0 | TestSuite: 1.1.0
```

Executando os Testes

Após salvar o arquivo com sua solução, execute o comando abaixo (em uma nova célula) para avaliar os testes automáticos:

```
TestSuite("EP01_01.extensão").run()
```

Substitua a extensão conforme a linguagem usada:

Linguagem	Extensão
Python	`.py`
Java	`.java`
C	`.c`
C++	`.cpp`
JavaScript	`.js`
R	`.r`

Como funciona: O `TestSuite` baixa os casos de teste do GitHub, executa seu programa com cada entrada e compara a saída com a esperada – calculando automaticamente sua nota.

⚠️ Importante: Regras e Boas Práticas

◆ Sobre a Entrada de Dados

O seu programa deve ler a entrada padrão (teclado).

- **Python:** Utilize `input()`.
- **Outras linguagens:** Utilize o comando de leitura padrão equivalente (`cin`, `Scanner`, etc.).

🔹 Configuração de IA no Colab

Para um melhor aprendizado, recomenda-se desativar o preenchimento automático de código por IA, pois não estará disponível durante as avaliações. Por exemplo no navegador Chrome:

- Vá em: **Ferramentas > Configurações > IA generativa**
- Desmarque: **Habilitar geração de código**

🔹 Integridade Acadêmica (Plágio)

Este recurso de testes locais aplica-se a EPs **sem variações**. No entanto:

- **Individualidade:** Cada aluno deve desenvolver sua própria solução.
- **Detecção de Similaridade:** O professor utiliza ferramentas que detectam cópias, **mesmo com alteração de nomes de variáveis ou espaços em branco**.

1.16.1 EP01_01 📏 Três métricas de distância em PDI

Nesta atividade, você deve escrever um programa que calcule as três distâncias mais usadas em PDI: **Euclidiana (L2)**, **City-Block (L1)** e ***Chessboard*** **(L∞)**.

- Leia **4 números reais** que representam as coordenadas: A_x, A_y, B_x, B_y.
- Calcule as três distâncias utilizando as fórmulas:

$$d_{\text{Euclidiana}} = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}$$

$$d_{\text{City-block}} = |B_x - A_x| + |B_y - A_y|$$

$$d_{\text{Chessboard}} = \max\left(|B_x - A_x|,\ |B_y - A_y|\right)$$

- Imprima os três resultados, cada um em uma linha, formatados com **duas casas decimais**, na ordem: **Euclidiana**, ***City-block***, ***Chessboard***.

📌 **Importante**:

- Utilize as funções matemáticas padrão da sua linguagem: `math.sqrt`, `abs` (ou `fabs`) e `max`.
- A saída deve conter **apenas os números** (um por linha), sem textos adicionais.
- Ver um simulador interativo para esta questão na **Figure 4.24** (gráfico com arrasto dos pontos e visualização das três métricas).

1.16.1.1 🖼️ Por que isso importa? – Custo computacional

Em uma imagem **1000×1000 pixels** (1 milhão de pixels), calcular a distância de cada pixel a um ponto de referência exige **1 milhão de operações**. A escolha da métrica afeta o desempenho:

Métrica	Operações por pixel	Custo relativo (1M pixels)	Quando usar
Euclidiana (L2)	2 subtrações, 2 multiplicações, 1 soma, 1 `sqrt`	🔴 Mais custosa – `sqrt` é cara	Distância "real" no espaço contínuo
***City-block* (L1)**	2 subtrações, 2 `abs`, 1 soma	🟡 Moderada – sem raiz quadrada	Grids, robótica, imagens binárias
***Chessboard* (L∞)**	2 subtrações, 2 `abs`, 1 `max`	🟢 Mais eficiente	Movimentos de peças, morfologia

A função `sqrt` é computacionalmente mais cara que operações como adição, subtração, multiplicação e valor absoluto. Em CPUs modernas, a diferença pode ser pequena (cerca de 1,5× a 3×), mas em sistemas embarcados ou em laços de milhões de iterações, qualquer ganho importa. Por isso, **quando o objetivo é apenas comparar distâncias** (ex.: encontrar o ponto mais próximo), use a **distância euclidiana ao quadrado**.

1.16.1.2 📋 Tarefa (especificação para VPL)

Entrada:
Uma única linha com quatro números reais: `Ax Ay Bx By`

Saída:
Três linhas, cada uma com um número real de duas casas decimais (Euclidiana, City-block, Chessboard).

1.16.1.3 📌 Exemplos

Entrada	Saída	Observação

`0`
`0`
`3`
`4`
	`5.00`
	`7.00`
	`4.00`
		Triângulo 3-4-5

`0`
`0`
`1`
`1`
	`1.41`
	`2.00`
	`1.00`
		Diagonal unitária

Exemplo de teste de `sqrt` em Python, com `timeit` isolando cada operação:

```python
import math
import timeit

N = 50_000_000

def apenas_soma():
    a, b = 3.0, 4.0
    return a + b

def soma_e_sqrt():
    a, b = 3.0, 4.0
    return math.sqrt(a*a + b*b)
```

```
t_soma = timeit.timeit(apenas_soma, number=N)
t_sqrt = timeit.timeit(soma_e_sqrt, number=N)

print(f"Soma simples       : {t_soma:.3f} s")
print(f"Soma + sqrt        : {t_sqrt:.3f} s")
print(f"Razão (sqrt/soma)  : {t_sqrt/t_soma:.2f}x")
```

```
Soma simples       : 4.883 s
Soma + sqrt        : 6.677 s
Razão (sqrt/soma)  : 1.37x
```

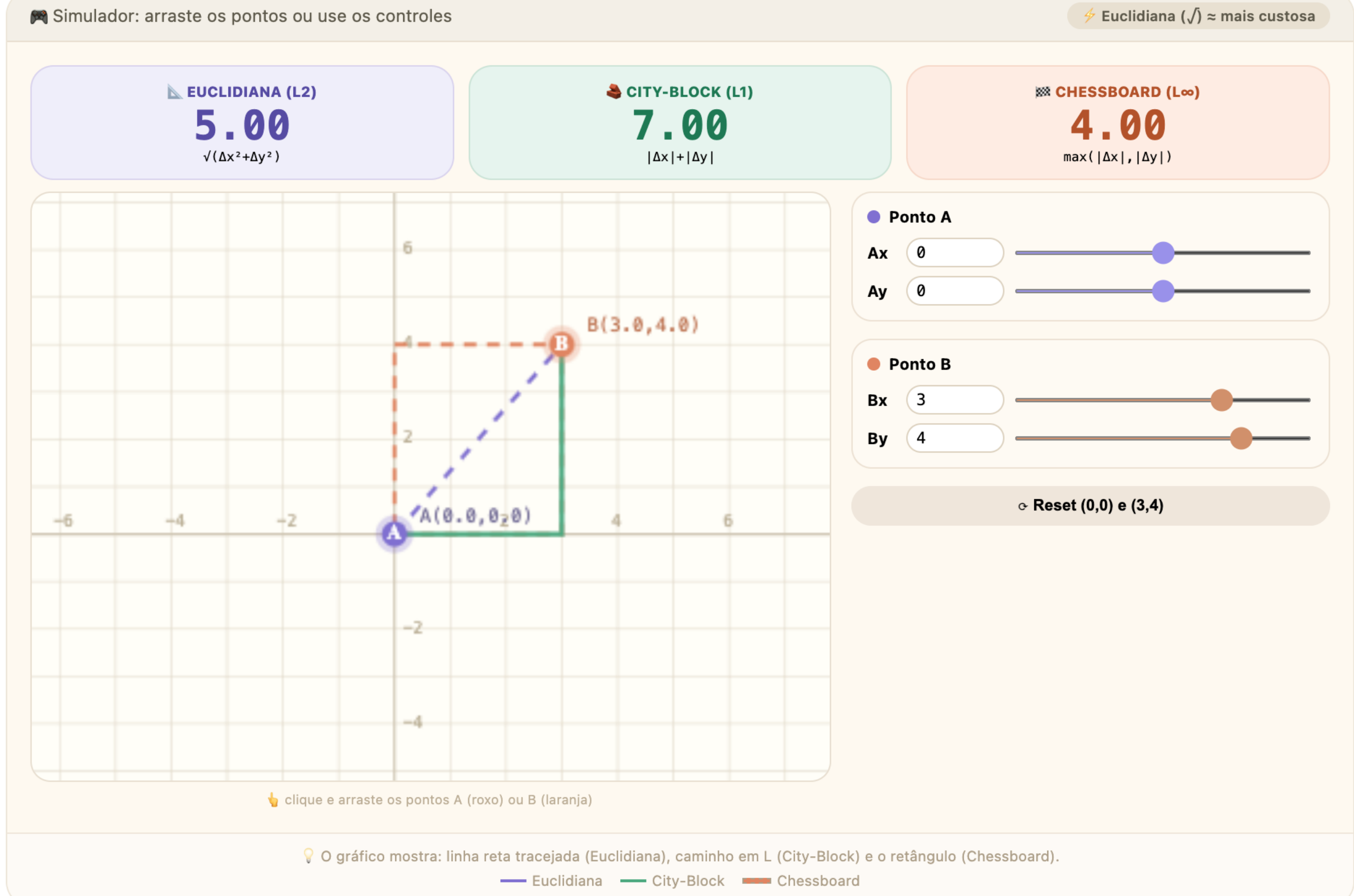


Figura 1.12: Simulador: Distâncias Euclidiana, *City-block* e *Chessboard*

1.16.1.4 Python

Basta criar uma célula de código normal e inserir o código Python. A entrada pode ser simulada usando `input()`, que funciona normalmente.

Exemplo de célula:

```
%%writefile EP01_01.py
# Código Python
x1,y1,x2,y2 = int(input()), int(input()), int(input()), int(input())
# Cálculo das diferenças
dx = abs(x2 - x1)
dy = abs(y2 - y1)

# 1. Distância Euclidiana (L2)
dist_euclidiana = (dx**2 + dy**2)**0.5
```

```
# 2. Distância City-block / Manhattan (L1)
dist_city_block = dx + dy

# 3. Distância Chessboard / Chebyshev (Linf)
dist_chessboard = max(dx, dy)

# Saída formatada conforme os casos de teste
print(f"{dist_euclidiana:.2f}")
print(f"{dist_city_block:.2f}")
print(f"{dist_chessboard:.2f}")
```

```
Writing EP01_01.py
```

```
# Espera que você digite 4 números inteiros ao executar esta célula.
# No Jupyter ou Google Colab, a mágica %run -i permite que o script leia do teclado.
# No terminal comum, você usaria: python3 EP01_01.py (sem o '!' e sem '%run').

# %run -i EP01_01.py
```

```
# Envia 4 inteiros como entrada padrão (stdin) para o script EP01_01.py usando um pipe
!echo -e "0\n0\n4\n4" | python3 EP01_01.py
```

```
5.66
8.00
4.00
```

```
TestSuite("EP01_01.py").run()
```

1.16.1.5 ☕ Java

Para executar Java no Colab, você precisa usar uma célula com o prefixo `%%writefile` para salvar o código em arquivo, compilar e executar.

```
%%writefile EP01_01.java
import java.util.Scanner;

class EP01_01 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);

        double x1 = s.nextDouble(), y1 = s.nextDouble();
        double x2 = s.nextDouble(), y2 = s.nextDouble();

        double dx = Math.abs(x2 - x1);
        double dy = Math.abs(y2 - y1);

        System.out.printf("%.2f\n", Math.sqrt(dx*dx + dy*dy));
        System.out.printf("%.2f\n", dx + dy);
        System.out.printf("%.2f\n", Math.max(dx, dy));
    }
}
```

```
Writing EP01_01.java
```

```
!javac EP01_01.java
!echo -e "0\n0\n4\n4" | java EP01_01
```

```
5,66
8,00
4,00
```

```
TestSuite("EP01_01.java").run()
```

1.16.1.6 C

De forma similar, use **`%%writefile`** para salvar o código, depois compile e execute. Para C, utilizamos o compilador **GCC**.

```
# Instalar o compilador GCC e ferramentas de build
# O build-essential inclui gcc, g++, make, etc.

import platform, subprocess
s = platform.system()
if s == "Linux":
    !sudo apt-get update -qq && sudo apt-get install -y build-essential -qq
elif s == "Darwin":
    if subprocess.run(["which","gcc"],capture_output=True).returncode != 0:
        print("  Instale o Xcode Command Line Tools: xcode-select --install")
else:
    print("  Windows: use WSL (recomendado) ou MinGW (https://www.mingw-w64.org)")
print("  Ambiente C/C++ pronto.")
```

```
  Ambiente C/C++ pronto.
```

```
%%writefile EP01_01.c
#include <stdio.h>
#include <math.h>

int main() {
    double x1, y1, x2, y2;
    if (scanf("%lf %lf %lf %lf", &x1, &y1, &x2, &y2) != 4) return 0;

    double dx = fabs(x2 - x1);
    double dy = fabs(y2 - y1);

    // Euclidiana, City-block e Chessboard
    printf("%.2f\n", sqrt(dx * dx + dy * dy));
    printf("%.2f\n", dx + dy);
    printf("%.2f\n", fmax(dx, dy));

    return 0;
}
```

```
Writing EP01_01.c
```

```
# Compila o arquivo .c gerando o executável EP01_01
# -lm é usado para linkar a biblioteca matemática (math.h) se necessário

!gcc EP01_01.c -o EP01_01 -lm
!echo -e "0\n0\n4\n4" | ./EP01_01
```

```
5.66
8.00
4.00
```

```
TestSuite("EP01_01.c").run()
```

1.16.1.7 💻 C++

De forma similar ao Java, use `%%writefile` para salvar o código, depois compile e execute. Lembre-se de que, no Colab, também é necessário instalar o seguinte:

```
# Instalar o compilador G++ para C++
# O build-essential inclui o g++, make, etc.

import platform, subprocess
s = platform.system()
if s == "Linux":
    !sudo apt-get update -qq && sudo apt-get install -y build-essential -qq
elif s == "Darwin" and subprocess.run(["which","g++"],capture_output=True).returncode != 0:
    print("  Instale o Xcode Command Line Tools: xcode-select --install")
elif s == "Windows":
    print("  Windows: use WSL (recomendado) ou MinGW (https://www.mingw-w64.org)")
print("  Compilador C++ pronto.")
```

```
  Compilador C++ pronto.
```

```
%%writefile EP01_01.cpp
#include <iostream>
#include <iomanip>
#include <cmath>
#include <algorithm>

int main() {
    double x1, y1, x2, y2;
    if (!(std::cin >> x1 >> y1 >> x2 >> y2)) return 0;

    double dx = std::abs(x2 - x1);
    double dy = std::abs(y2 - y1);

    std::cout << std::fixed << std::setprecision(2);

    // Euclidiana, City-block e Chessboard
    std::cout << std::sqrt(dx*dx + dy*dy) << std::endl;
    std::cout << (dx + dy) << std::endl;
    std::cout << std::max(dx, dy) << std::endl;

    return 0;
}
```

```
Writing EP01_01.cpp
```

```
!g++ EP01_01.cpp -o EP01_01
!echo -e "0\n0\n4\n4" | ./EP01_01
```

```
5.66
8.00
4.00
```

```
TestSuite("EP01_01.cpp").run()
```

1.16.1.8 🌐 JavaScript (Node.js)

Para JavaScript, use `%%writefile` para criar o arquivo e execute com Node:

```
%%writefile EP01_01.js
function escreva(s) { try { document.write(s + "<br>"); } catch(e) { console.log(s); } }

process.stdin.once('data', data => {
  const valores = data.toString().trim().split(/\s+/);
  const x1 = parseFloat(valores[0]);
  const y1 = parseFloat(valores[1]);
  const x2 = parseFloat(valores[2]);
  const y2 = parseFloat(valores[3]);

  const dx = Math.abs(x2 - x1);
  const dy = Math.abs(y2 - y1);

  // Euclidiana, City-block e Chessboard
  escreva(Math.sqrt(dx * dx + dy * dy).toFixed(2));
  escreva((dx + dy).toFixed(2));
  escreva(Math.max(dx, dy).toFixed(2));
});
```

```
Writing EP01_01.js
```

```
TestSuite("EP01_01.js").run()
```

1.16.1.9 📊 R

No Colab, pode-se executar código R diretamente utilizando a mágica `%%R`.
O programa deve ler **quatro números** (x1, y1, x2, y2) e exibir a distância euclidiana com **duas casas decimais**.

Exemplo de célula:

```
%%R
dados <- scan(file = "stdin", n = 4, quiet = TRUE)
x1 <- dados[1]; y1 <- dados[2]; x2 <- dados[3]; y2 <- dados[4]
dist <- sqrt((x2 - x1)^2 + (y2 - y1)^2)
cat(sprintf("%.2f", dist))
```

```
# Instalar o R e o Rscript
# O r-base instala o ambiente R completo, incluindo o Rscript

import platform, subprocess
s = platform.system()
if s == "Linux":
    !sudo apt-get update -qq && sudo apt-get install -y r-base -qq
elif s == "Darwin":
    if subprocess.run(["which","R"],capture_output=True).returncode != 0:
        print("  Instale o R no macOS: https://cran.r-project.org/bin/macosx/")
else:
    print("  Windows: baixe o R em https://cran.r-project.org/bin/windows/base/")
print("  Ambiente R pronto.")
```

```
Ambiente R pronto.
```

Para testar da mesma forma que nos exemplos anteriores, deve-se usar a entrada padrão (stdin) no terminal ou adaptar o código conforme abaixo:

```
%%writefile EP01_01.r
dados <- scan(file = "stdin", n = 4, quiet = TRUE)
dx <- abs(dados[3] - dados[1])
dy <- abs(dados[4] - dados[2])

# Saídas: Euclidiana, City-block e Chessboard
cat(sprintf("%.2f\n%.2f\n%.2f\n",
    sqrt(dx^2 + dy^2),
    dx + dy,
    max(dx, dy)))
```

```
Writing EP01_01.r
```

```
!echo -e "0\n0\n4\n4" | Rscript EP01_01.r
```

```
5.66
8.00
4.00
```

```
TestSuite("EP01_01.r").run()
```

1.16.2 EP01_02 📊 Desempenho Preditivo — Métricas de ML em VC

Nesta atividade, você entrará no mundo do **Aprendizado de Máquina** (*Machine Learning*). Seu objetivo é avaliar o desempenho de um classificador binário calculando métricas a partir de uma **Matriz de Confusão**.

1.16.2.1 🧠 Por que a métrica certa importa?

Imagine um detector de **moedas de R$ 1,00**. O impacto do erro define a métrica prioritária:

Métrica	Exemplo Prático	Importância em PDI/VC
Acurácia	**Contagem de Grãos**	Útil quando as classes são equilibradas (ex: metade dos grãos com defeito, metade saudáveis).
Precisão	**Segurança/Biometria**	Crucial para evitar **Falsos Positivos** (ex: não permitir que um impostor acesse um sistema por erro de reconhecimento).
Sensibilidade	**Saúde (Tumores)**	Crucial para evitar **Falsos Negativos** (ex: não deixar um tumor passar despercebido em um exame de Raio-X).
F1-score	**Cédulas de Dinheiro**	Ideal para um **equilíbrio** entre não rejeitar notas verdadeiras e não aceitar notas falsas.

1.16.2.2 📊 A Matriz de Confusão

	Predita Positiva	**Predita Negativa**
Real Positiva	**VP** (Verdadeiro Positivo)	**FN** (Falso Negativo)
Real Negativa	**FP** (Falso Positivo)	**VN** (Verdadeiro Negativo)

Tarefa:

1. Leia **4 valores inteiros** na ordem: `VP`, `FN`, `FP`, `VN`.
2. Calcule as métricas utilizando as fórmulas:

$$\text{Acurácia} = \frac{VP + VN}{VP + VN + FP + FN}$$

$$\text{Precisão} = \frac{VP}{VP + FP}$$

$$\text{Sensibilidade (Recall)} = \frac{VP}{VP + FN}$$

$$\text{F1-score} = \frac{2 \times \text{Precisão} \times \text{Sensibilidade}}{\text{Precisão} + \text{Sensibilidade}}$$

3. Imprima os resultados formatados com **duas casas decimais**, um por linha.

📌 **Importante**:

- Use divisão em ponto flutuante para evitar resultados truncados.
- A ordem de saída deve ser: **Acurácia**, **Precisão**, **Sensibilidade** e **F1-score**.
- Veja a simulação interativa deste EP na Figure 1.13.

1.16.2.3 📌 Exemplo de Execução

Entrada	Saída	Observação
`40`	`0.75`	Acurácia
`10`	`0.73`	Precisão
`15`	`0.80`	Sensibilidade
`35`	`0.76`	F1-score

(Nota: VP=40, FN=10, FP=15, VN=35. Total de casos = 100)

```
%%writefile EP01_02.py
# sua solução
```

```
Writing EP01_02.py
```

```
TestSuite("EP01_02.py").run()
```

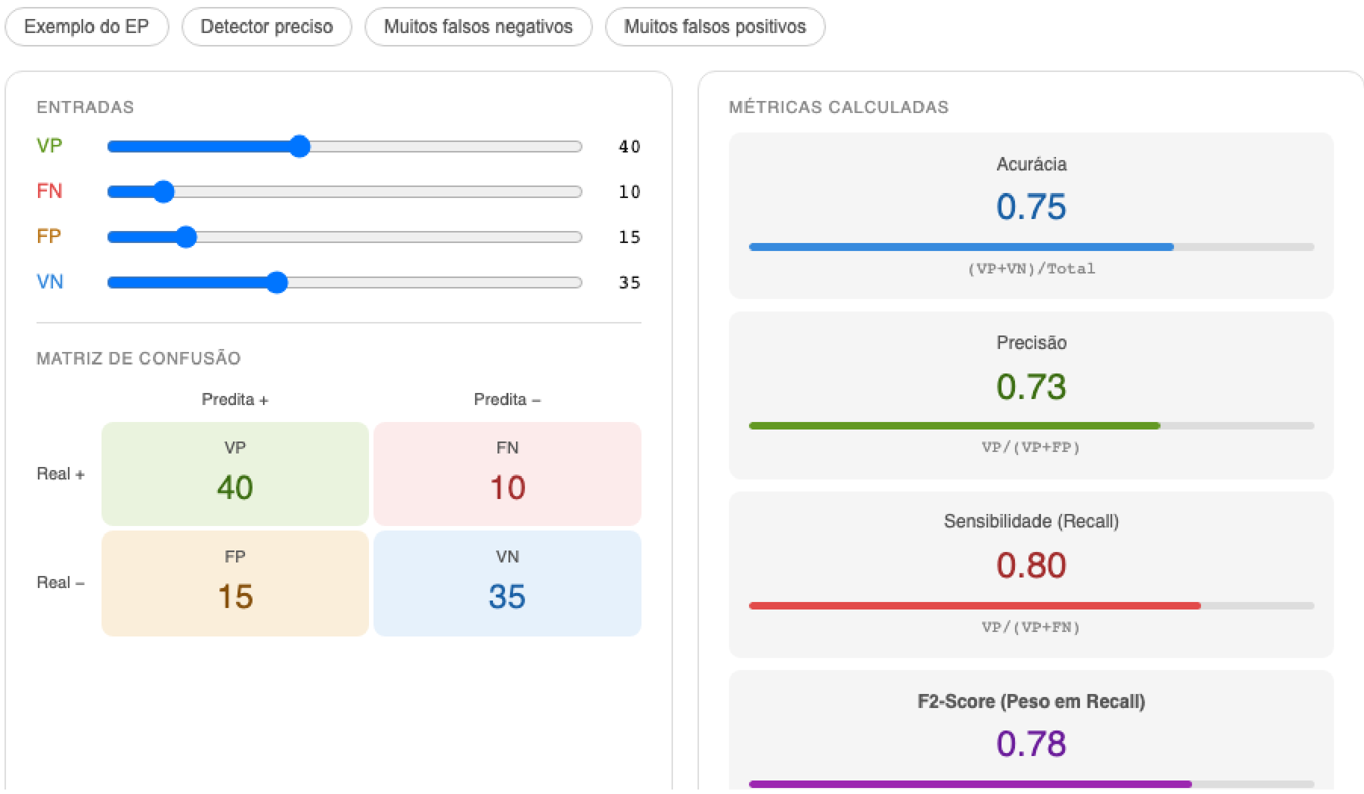


Figura 1.13: Simulador: Desempenho Preditivo — Métricas de ML

1.16.3 EP01_03 ↗ *Mean Average Precision* (mAP) — Curva Precisão-Sensibilidade

Nesta atividade, você avaliará um classificador binário (ex.: detecção de desmatamento em imagens de satélite) através da **curva Precisão-Sensibilidade** e da métrica **mAP (*Mean Average Precision*)**. O mAP é padrão em competições como COCO (*Common Objects in Context*) e PASCAL VOC (*Visual Object Classes*) e dos modelos YOLO (*You Only Look Once*).

1.16.3.1 🧠 Por que o mAP é a métrica-padrão?

Na EP01_02 você viu que a escolha do limiar altera significativamente a Precisão e a Sensibilidade. O **mAP (Mean Average Precision)** resolve isso: avalia o modelo em **vários limiares** (cada limiar deve gerar uma matriz de confusão diferente) e resume o desempenho pela **área sob a curva Precisão-Sensibilidade (P-S)**.

Enquanto o F1-*Score* olha para um único ponto de equilíbrio, o mAP considera a curva inteira. Quanto mais próximo de **1,0**, melhor o detector em todos os limiares e classes (ex.: moedas de 25, 50 e 1 real).

Métrica	O que resume	Limitação
F1-*Score*	Equilíbrio P × S num único limiar	Depende do limiar escolhido
AP	Área sob a curva P-S de uma classe	Válida apenas para uma classe
mAP	Média das APs sobre todas as classes	Mais complexo de implementar

Referências: Roboflow — mAP · Vídeo explicativo

1.16.3.2 Como o mAP é calculado — passo a passo

1. **Limiares fixos** (use sempre esta lista):

```
limiares = [0.00, 0.09, 0.21, 0.31, 0.39, 0.52, 0.60, 0.71, 0.81, 0.89, 1.00]
```

2. Para cada limiar (t), classifique as amostras: `predito = 1 se confiança  t, senão 0`. Calcule VP, FP, FN, VN e obtenha Precisão((t)) e Sensibilidade((t)).
3. Monte a curva P-S: pares (Sensibilidade((t)), Precisão((t))), ordenados por Sensibilidade crescente.
4. **Monotonize a Precisão**:

$$P_{\text{mono}}[i] = \max_{j \geq i} P[j]$$

5. Calcule a **AP** (área sob a curva monotônica) usando a **regra do trapézio** (aproximação mais precisa que a simples soma de Riemann):

$$AP = \sum_{i=1}^{m-1} \frac{P_{\text{mono}}[i-1] + P_{\text{mono}}[i]}{2} \cdot (S[i] - S[i-1])$$

6. **mAP** = média das APs de todas as classes. Neste EP há apenas **1 classe**, portanto mAP = AP.

> ℹ Note
>
> 📐 **Diferença resumida:**
> A *soma de Riemann* aproxima a área por retângulos, podendo subestimar ou superestimar. A **regra do trapézio** usa trapézios, reduzindo o erro ao considerar a média entre os valores nos extremos do intervalo, sendo geralmente mais precisa para funções suaves por partes, como a curva Precisão-Sensibilidade.

1.16.3.3 📋 Tarefa

Leia um inteiro `n` (quantidade de amostras). Em seguida leia `n` linhas, cada uma com: `verdade` (0 ou 1) e `confiança` (float 0.0–1.0).

Calcule e imprima, para o **limiar 0.85** (índice 9 da lista):

- Matriz de Confusão (VP, FN, FP, VN)
- Acurácia, Precisão, Sensibilidade e F1-Score

Em seguida, para **todos os limiares**, imprima:

- Precisões cruas, Precisões monotônicas e Sensibilidades, separadas por `,`
- mAP final

1.16.3.4 📌 Importante

- Limiar fixo para as métricas individuais: **0.85**
- Divisão segura: se denominador for zero, use 0
- Formatação: duas casas decimais
- Monotonize de trás para frente
- A Figure 1.14 apresenta uma simulação desta questão

1.16.3.5 📌 Exemplo de Execução

Entrada	Saída Esperada
7 0 0.94 1 0.80 1 0.69 0 0.67 1 0.30 1 0.15 1 0.15	# MÉTRICAS PARA O LIMIAR 0.85 # Matriz de Confusão: VP = 0, FN = 5 FP = 1, VN = 1 Métricas de Avaliação: Acurácia: 0.14 Precisão: 0.00 Sensibilidade: 0.00 F1-Score: 0.00 # MÉTRICAS PARA TODOS OS LIMIARES # Precisões: 0.00, 0.00, 0.00, 0.50, 0.50, 0.50, 0.50, 0.50, 0.60, 0.71, 0.71 Precisões mon.: 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71 Sensibilidades: 0.00, 0.00, 0.00, 0.20, 0.40, 0.40, 0.40, 0.40, 0.60, 1.00, 1.00 mAP: 0.71

1.16.3.6 Dica para calcular o AP (com regra do trapézio)

```
def calcular_AP(verdades, confiancas, limiares):
    m = len(limiares)
    precisoes = [0.0] * m
    sensibilidades = [0.0] * m
    for i in range(m):
        p, s = calcular_metricas(verdades, confiancas, limiares[i])
        precisoes[m-1-i] = p
        sensibilidades[m-1-i] = s
    prec_mono = precisoes.copy()
    for i in range(m-2, -1, -1):
        if prec_mono[i] < prec_mono[i+1]:
            prec_mono[i] = prec_mono[i+1]
    AP = 0.0
    for i in range(1, m):
        # Regra do trapézio: média das alturas vezes a base
        area_trapezio = (prec_mono[i-1] + prec_mono[i]) / 2.0
        AP += area_trapezio * (sensibilidades[i] - sensibilidades[i-1])
    return precisoes, prec_mono, sensibilidades, AP
```

```
%%writefile EP01_03.py
# sua solução
```

```
Writing EP01_03.py
```

```
TestSuite("EP01_03.py").run()
```

1.16.4 EP01_04 Leitura e Informações de uma Imagem em Matriz

Nesta atividade, você deve escrever um programa que processe uma imagem digital representada como uma matriz de pixels em tons de cinza.

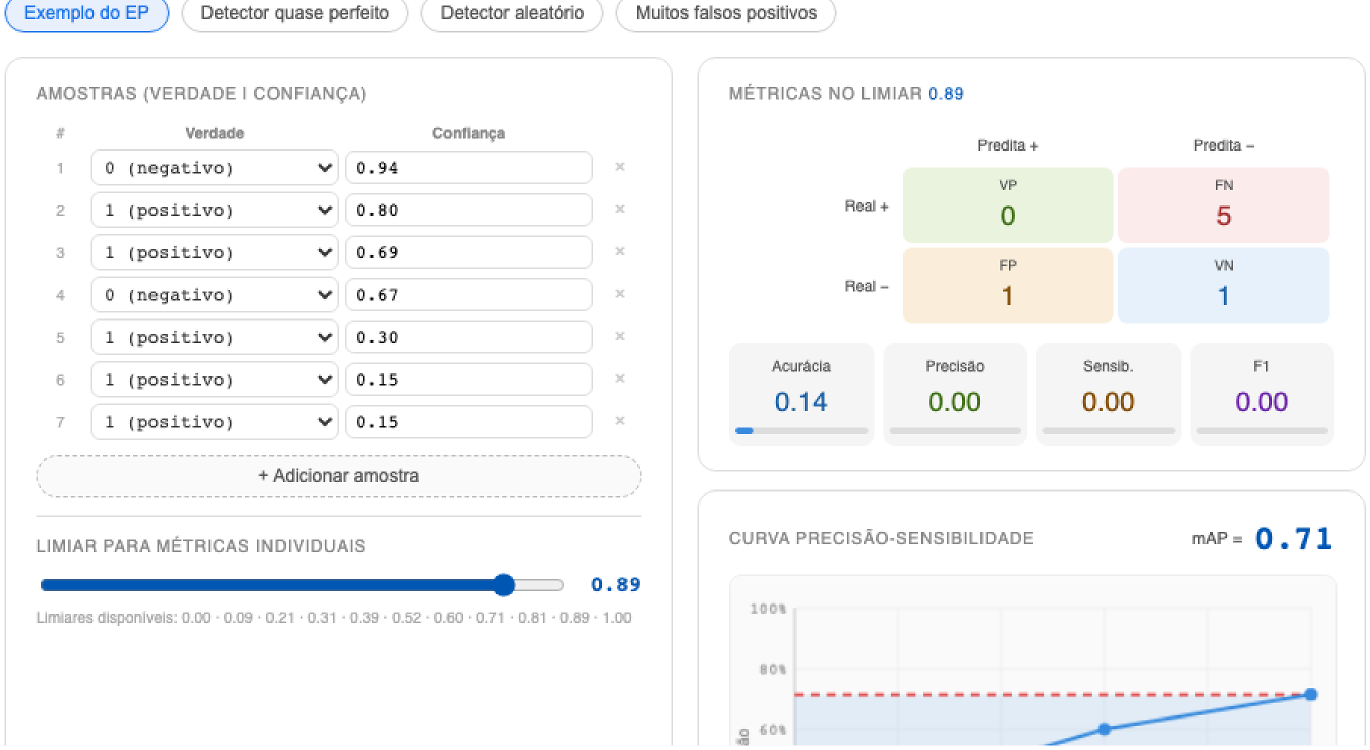


Figura 1.14: Simulador: *Mean Average Precision* (mAP) — Curva P-S

- Leia dois inteiros **L** e **C**, representando o número de linhas e colunas.
- Leia os **L * C** valores inteiros que compõem a matriz da imagem (cada valor entre 0 e 255).
- Calcule e imprima as seguintes informações:

1. O número de linhas.
2. O número de colunas.
3. O valor do maior pixel (**Máximo**).
4. O valor do menor pixel (**Mínimo**).
5. A **Média** aritmética de todos os pixels.

📌 **Importante**:

- A saída deve seguir exatamente o formato rotulado (ex: `Linhas: X`).
- O valor da média deve ser formatado com **duas casas decimais**.
- Ver um simulador interativo para esta questão na **Figure 1.15** (grade interativa para visualização de intensidades e cálculos em tempo real).

1.16.4.1 🧠 Por que isso importa? – A Imagem como Dados

Toda imagem digital é, no fundo, uma estrutura de dados. Em tons de cinza de 8 bits, cada pixel é um valor escalar. Extrair estatísticas básicas é o primeiro passo para:

Operação	Utilidade Prática
Máximo/Mínimo	Identificar se a imagem está "lavada" (baixo contraste) ou saturada.

Operação	Utilidade Prática
Média	Calcular o brilho global da cena para ajustes de exposição.
Normalização	Redimensionar os valores para intervalos como $[0, 1]$ em redes neurais.

1.16.4.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém o inteiro `L` (linhas).

A segunda linha contém o inteiro `C` (colunas).

As linhas seguintes contêm os elementos da matriz.

Saída:

Cinco linhas formatadas conforme o exemplo:

`Linhas: L`

`Colunas: C`

`Max: V`

`Min: V`

`Media: V.VV`

1.16.4.3 📌 Exemplos

Entrada	Saída	Observação
2 3 0 128 255 50 100 200	Linhas: 2 Colunas: 3 Max: 255 Min: 0 Media: 122.17	Imagem pequena de alto contraste

```
%%writefile EP01_04.py
# sua solução
```

```
Writing EP01_04.py
```

```
TestSuite("EP01_04.py").run()
```

1.16.5 EP01_05 🔄 Negativo de uma Imagem em Tons de Cinza

Nesta atividade, deve-se escrever um programa que calcule o negativo de uma imagem digital.

- Leia dois inteiros **L** e **C**, representando o número de linhas e colunas.

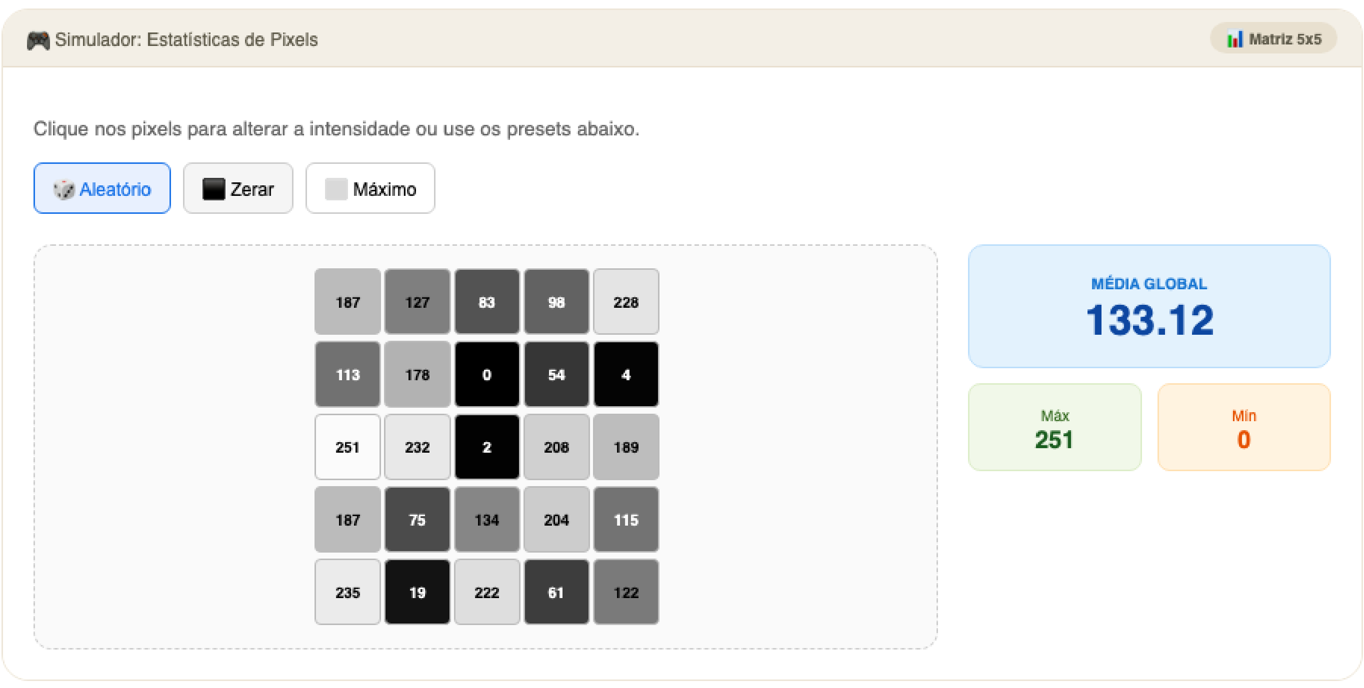


Figura 1.15: Simulador: Estatísticas de Pixels

- Leia os valores inteiros que compõem a matriz da imagem.
- Para cada pixel, aplique a transformação de inversão:

$$pixel_{negativo} = 255 - pixel_{original}$$

- Imprima a matriz resultante, mantendo o formato original (L linhas e C colunas).

📌 **Importante**:

- Os valores de cada linha na saída devem ser separados por um **espaço em branco**.
- A saída deve conter **apenas os números** da matriz resultante.
- Ver um simulador interativo para esta questão na **Figure 1.16** (comparação em tempo real entre a matriz original e seu negativo).

1.16.5.1 🧠 Por que isso importa? – Inversão de Intensidade

O **negativo** é uma transformação linear básica que inverte a escala de brilho. É uma ferramenta essencial para o olho humano identificar detalhes claros que estão "escondidos" em fundos mais escuros, sendo amplamente utilizada em:

Aplicação	Utilidade
Imagens Médicas	Melhora a visualização de anomalias em tecidos densos (ex: Raios-X).
Astronomia	Destacar galáxias e nebulosas tênues contra o vazio do espaço.

Aplicação	Utilidade
Artes Digitais	Efeitos estéticos e preparação de máscaras de seleção.

1.16.5.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém o inteiro `L`.

A segunda linha contém o inteiro `C`.

As linhas seguintes contêm os elementos da matriz.

Saída:

A matriz invertida com `L` linhas e `C` colunas.

1.16.5.3 📌 Exemplos

Entrada	Saída	Observação
2 3 0 128 255 50 100 255	255 127 0 205 155 55	Onde era 0 (preto) vira 255 (branco)

```
%%writefile EP01_05.py
# sua solução
```

```
Writing EP01_05.py
```

```
TestSuite("EP01_05.py").run()
```

1.16.6 EP01_06 🎨 Conversão RGB → Tons de Cinza (ITU-R BT.601)

Nesta atividade, você deve escrever um programa que converta pixels coloridos (RGB) para tons de cinza utilizando a ponderação fisiológica do padrão ITU-R BT.601.

- Leia dois inteiros **L** e **C**, representando o número de linhas e colunas.
- Leia **L** × **C** triplas de inteiros, onde cada tripla representa os canais **R** (Red), **G** (Green) e **B** (Blue) de um pixel.
- Para cada pixel, calcule o valor de cinza (g) usando a fórmula:

$$g = \text{round}(0.299 \times R + 0.587 \times G + 0.114 \times B)$$

- Imprima a matriz resultante (L linhas e C colunas) contendo os valores inteiros convertidos.

📌 **Importante**:

- Utilize a função `round()` da sua linguagem para garantir o arredondamento correto para o inteiro mais próximo.

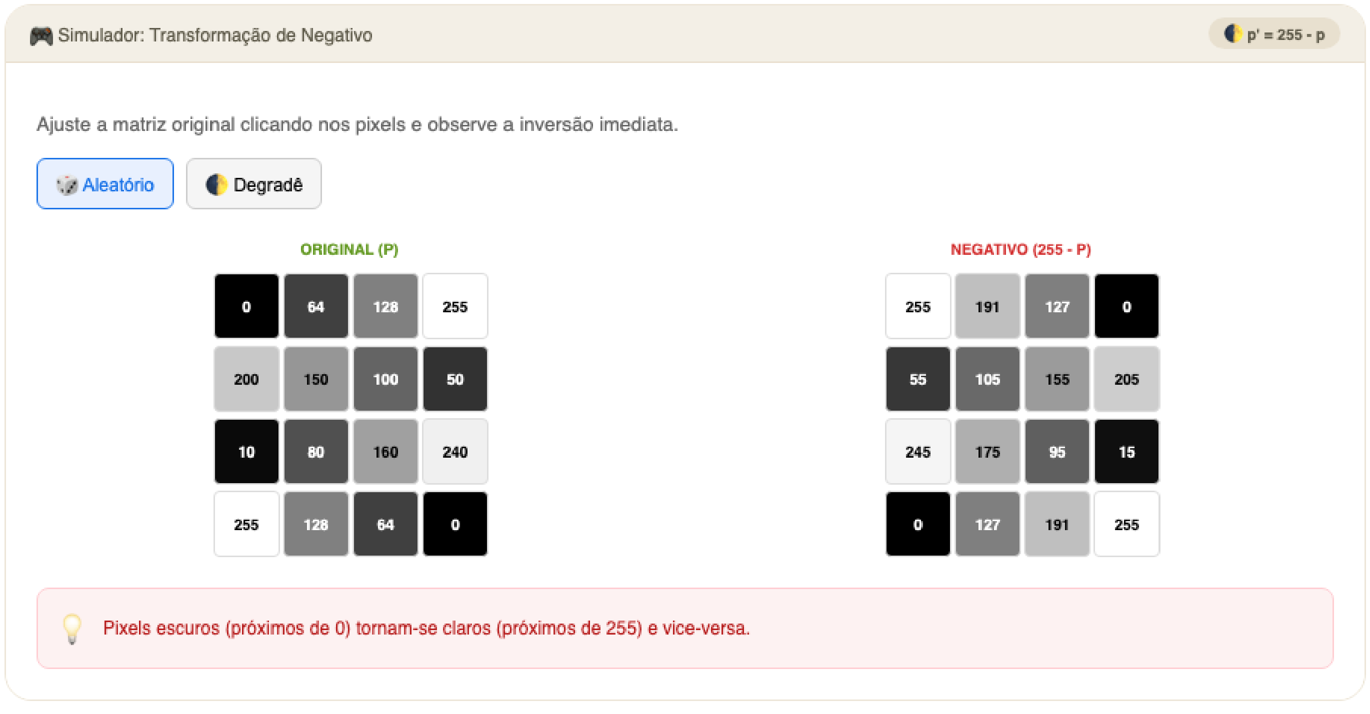


Figura 1.16: Simulador: Transformação de Negativo

- A saída deve conter apenas os valores de cinza, mantendo a estrutura de matriz (separados por espaço na linha).
- Ver um simulador interativo para esta questão na **Figure 1.17** (ajuste os sliders para ver como cada cor contribui para o brilho final).

1.16.6.1 🧠 Por que não usar apenas a média?

O olho humano não percebe todas as cores com a mesma intensidade. Somos muito mais sensíveis ao **Verde** do que ao **Azul** devido à nossa evolução biológica. O padrão ITU-R BT.601 utiliza pesos específicos para criar uma imagem em cinza que pareça naturalmente correta para nossa visão:

Canal	Peso	Percepção Humana
🟢 **Verde**	**58.7%**	Máxima sensibilidade (distinção de folhagens).
🔴 **Vermelho**	**29.9%**	Sensibilidade média.
🔵 **Azul**	**11.4%**	Baixa sensibilidade (tons mais escuros).

1.16.6.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém o inteiro `L`.

A segunda linha contém o inteiro `C`.

As linhas seguintes contêm triplas de inteiros `R G B` para cada pixel.

Saída:

A matriz de cinzas com `L` linhas e `C` colunas.

1.16.6.3 📌 Exemplos

Entrada	Saída	Observação
1 3 255 0 0 0 255 0 0 0 255	76 150 29	Note como o Verde (150) é mais brilhante que o Azul (29)

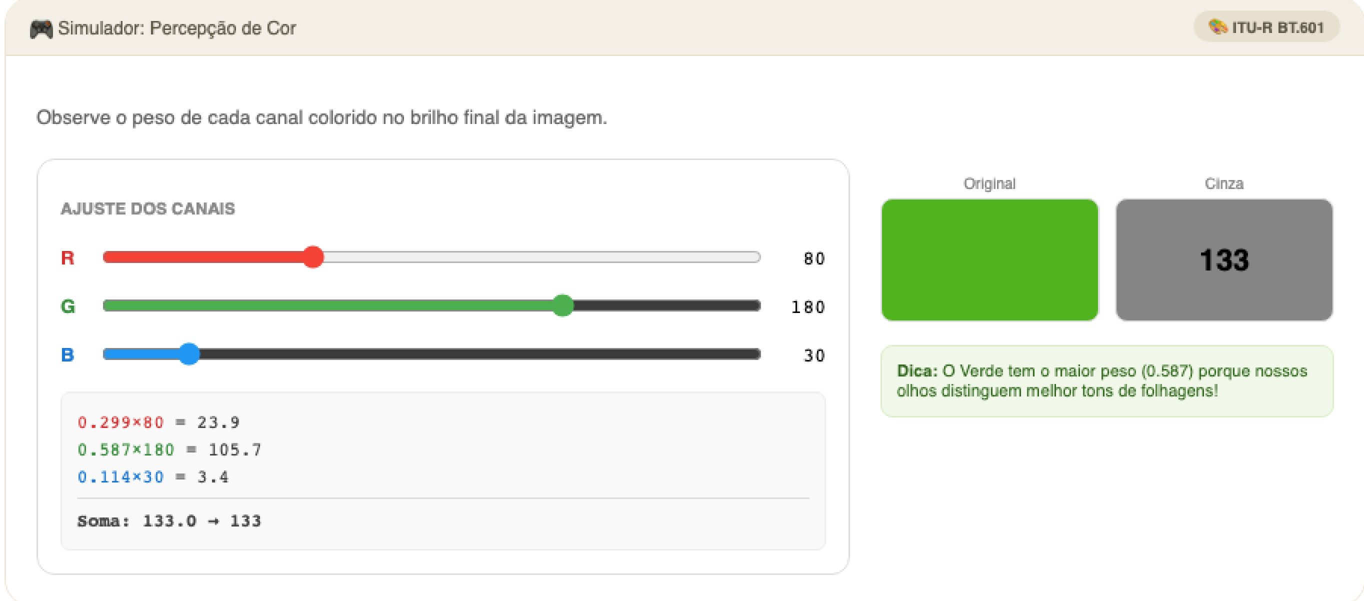


Figura 1.17: Simulador: Percepção de Cor e Pesos ITU-R

```
%%writefile EP01_06.py
# sua solução
```

```
Writing EP01_06.py
```

```
TestSuite("EP01_06.py").run()
```

1.16.7 EP01_07 ⚫ Limiarização Manual: Imagem Binária

Nesta atividade, você deve escrever um programa que realize a segmentação de uma imagem através da limiarização (*thresholding*).

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz.
- Leia um inteiro **T**, que será o valor do limiar (corte).
- Leia os valores inteiros da matriz.
- Para cada pixel p, aplique a seguinte regra de binarização:

$$\text{resultado} = \begin{cases} 255 & \text{se } p > T \\ 0 & \text{se } p \leq T \end{cases}$$

- Imprima a matriz resultante contendo apenas os valores 0 ou 255.

📌 **Importante**:

- Preste atenção ao operador: o pixel só vira branco (255) se for **estritamente maior** que T.
- A saída deve manter a estrutura de matriz (L linhas e C colunas).
- Ver um simulador interativo para esta questão na **Figure 1.18** (ajuste o slider de T para observar como os objetos são isolados do fundo).

1.16.7.1 🧠 O que é Segmentação?

A limiarização é o método mais simples de separar objetos de interesse do fundo da imagem. Ao transformar tons de cinza em preto e branco puro, criamos um mapa binário que facilita a contagem de objetos ou a identificação de formas:

Valor do Pixel (p)	Condição	Resultado Final
Escuro ($p \leq T$)	Fundo/Ruído	**0** (Preto)
Claro ($p > T$)	Objeto/Destaque	**255** (Branco)

1.16.7.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém o inteiro `L`.

A segunda linha contém o inteiro `C`.

A terceira linha contém o inteiro `T` (limiar).

As linhas seguintes contêm os elementos da matriz.

Saída:

A matriz binarizada (0 ou 255) com `L` linhas e `C` colunas.

1.16.7.3 📌 Exemplos

Entrada	Saída	Observação
2 4 128 0 100 128 200 50 129 255 64	0 0 0 255 0 255 255 0	Note que o valor 128 virou 0 (pois $128 \leq 128$)

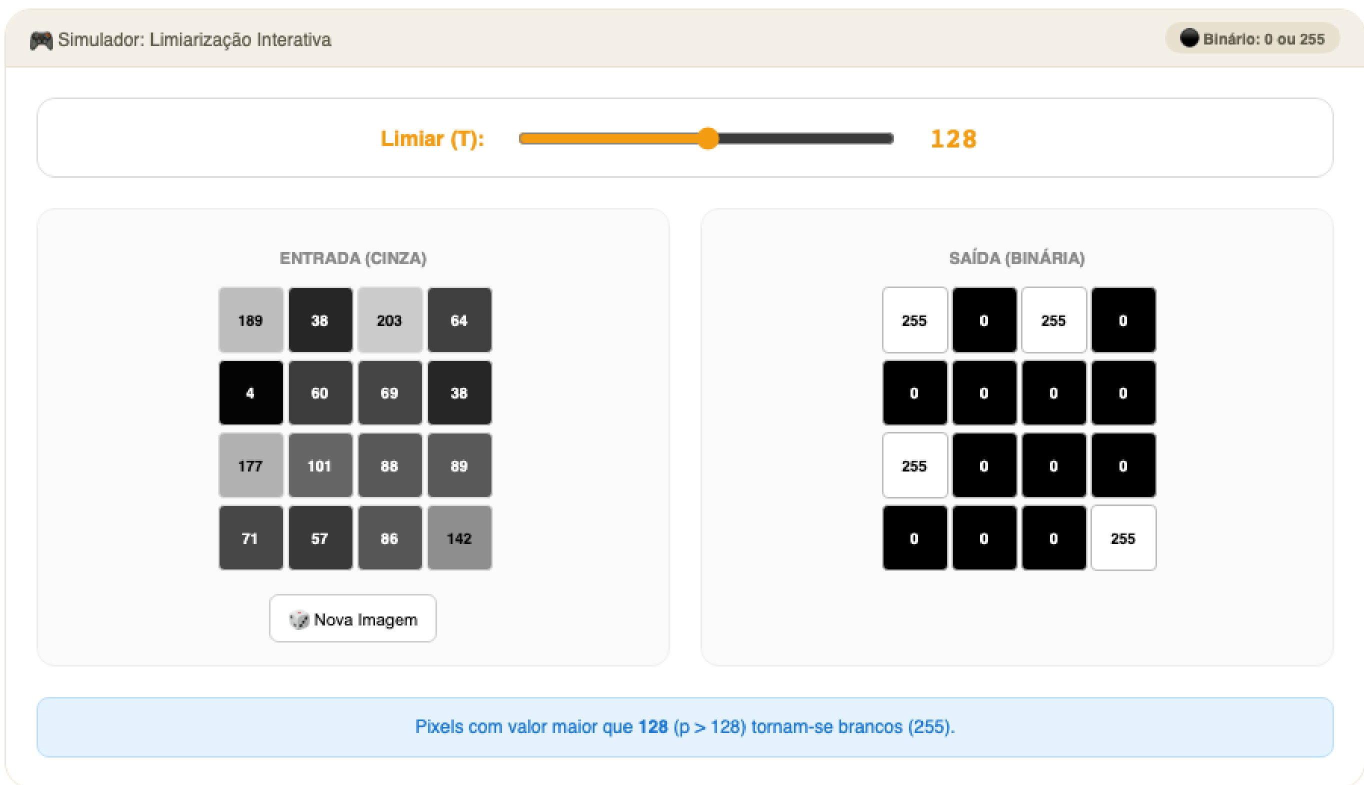


Figura 1.18: Simulador: Limiarização Interativa

```
%%writefile EP01_07.py
# sua solução
```

```
Writing EP01_07.py
```

```
TestSuite("EP01_07.py").run()
```

1.16.8 EP01_08 🎨 Remapeamento por Faixa de Intensidade

Nesta atividade, você deve escrever um programa que aplique transformações lineares distintas a diferentes regiões de intensidade da imagem.

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz.
- Leia os inteiros **T** (limiar), (delta 1) e (delta 2).
- Leia os valores inteiros da matriz.
- Para cada pixel p, aplique a regra de remapeamento condicional:

$$\text{resultado} = \begin{cases} p + \delta_1 & \text{se } p < T \\ p + \delta_2 & \text{se } p \geq T \end{cases}$$

- Imprima a matriz resultante com os novos valores de intensidade.

📌 **Importante**:

- é o offset aplicado aos pixels escuros (abaixo do limiar).
- é o offset aplicado aos pixels claros (maior ou igual ao limiar).
- Os casos de teste garantem que o resultado estará sempre no intervalo válido de **0 a 255**, portanto, não é necessário tratar saturação ou arredondamentos.
- A saída deve manter a estrutura de matriz (**L** linhas e **C** colunas).

1.16.8.1 🧠 Transformação Condicional de Pixels

Em Processamento Digital de Imagens (PDI), frequentemente precisamos tratar regiões de forma independente. Esta técnica permite, por exemplo, clarear apenas as sombras de uma fotografia (aumentando os pixels escuros) sem estourar o brilho das áreas já claras, ou vice-versa.

Faixa de Intensidade	Condição	Operação
Pixels Escuros	$p < T$	$p + \delta_1$
Pixels Claros	$p \geq T$	$p + \delta_2$

1.16.8.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém os inteiros `L` e `C`.

A segunda linha contém os inteiros `T`, e .

As linhas seguintes contêm os elementos da matriz.

Saída:

A matriz transformada com `L` linhas e `C` colunas, com valores separados por espaço.

1.16.8.3 📌 Exemplos

Entrada	Saída	Observação

`2 4`
`128 60 -40`
`0 100 150 255`
`80 128 200 30`
	`60 160 110 215`
	`140 88 160 90`
		Pixels $<$ 128 somam 60. Pixels 128 subtraem 40.

```
%%writefile EP01_08.py
# sua solução
```

```
Writing EP01_08.py
```

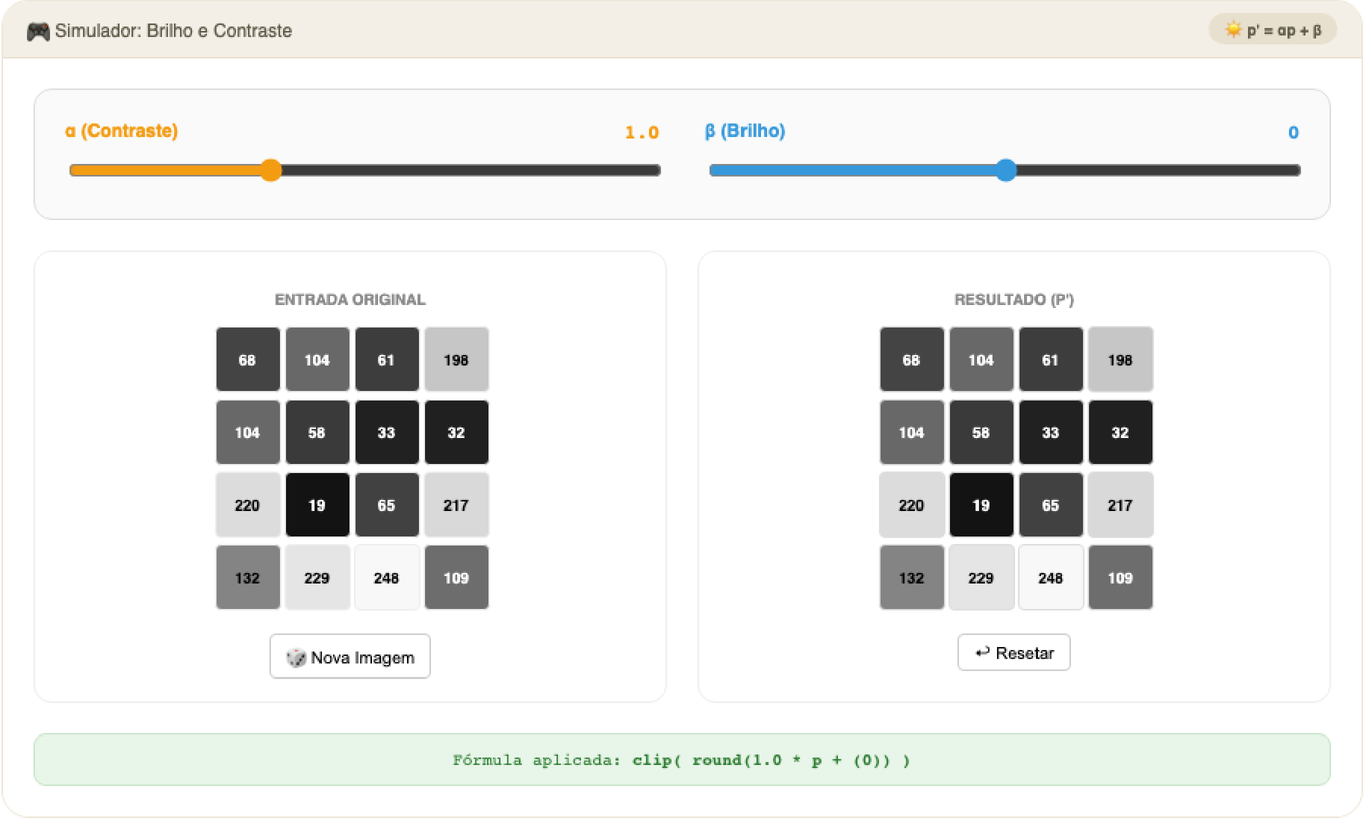


Figura 1.19: Simulador: Ajuste de Brilho e Contraste

```
TestSuite("EP01_08.py").run()
```

1.16.9 EP01_09 🏁 Padrão de Tabuleiro: Matriz de Xadrez

Nesta atividade, você deve escrever um programa que gere uma imagem sintética no padrão de um tabuleiro de xadrez.

- Leia dois inteiros **L** (linhas) e **C** (colunas).
- Gere uma matriz onde os valores alternam entre **0** (preto) e **1** (branco).
- A lógica de preenchimento deve seguir a regra da paridade:
- O elemento na posição $(0, 0)$ é sempre **0**.
- Um pixel na posição (i, j) será **1** se a soma dos índices $(i + j)$ for **ímpar**.
- Um pixel na posição (i, j) será **0** se a soma dos índices $(i + j)$ for **par**.

📌 **Importante**:

- As cores devem alternar corretamente tanto horizontalmente quanto verticalmente.
- A saída deve ser a matriz impressa linha por linha, com os elementos separados por um espaço.
- Ver um simulador interativo para esta questão na **Figure 1.20** (ajuste as dimensões para visualizar a construção da malha e a saída textual correspondente).

1.16.9.1 🧠 Padrões Sintéticos

Criar padrões geométricos é um exercício fundamental para dominar a **lógica de índices** em matrizes. Em Processamento Digital de Imagens, o padrão de xadrez não é apenas estético; ele é amplamente utilizado para:

Aplicação	Utilidade
Calibração de Câmera	Estimar parâmetros intrínsecos e extrínsecos da lente.
Correção de Distorção	Identificar e corrigir o efeito de "barril" ou "almofada" em lentes grande-angulares.
Mapeamento 3D	Projetar padrões conhecidos para reconstruir superfícies em sistemas de luz estruturada.

1.16.9.2 📋 Tarefa (especificação para VPL)

Entrada:

Uma linha contendo o inteiro `L` (linhas).

Uma linha contendo o inteiro `C` (colunas).

Saída:

A matriz de xadrez com `L` linhas e `C` colunas, impressa com espaços entre os elementos.

1.16.9.3 📌 Exemplos

Entrada	Saída	Observação
3 4	0 1 0 1 1 0 1 0 0 1 0 1	Note que cada linha começa com o inverso da anterior

```
%%writefile EP01_09.py
# sua solução
```

```
Writing EP01_09.py
```

```
TestSuite("EP01_09.py").run()
```

1.16.10 EP01_10 📄 Metadados: Leitura de Arquivo PGM

Nesta atividade, você deve ler um arquivo de imagem no formato **PGM (Portable Gray Map)** e extrair suas dimensões a partir do cabeçalho.

- O formato **PGM (P2)** é um arquivo de texto simples (ASCII) que armazena imagens em tons de cinza.
- O arquivo possui um **cabeçalho** estruturado da seguinte forma:

1. **Versão:** O identificador `P2`.

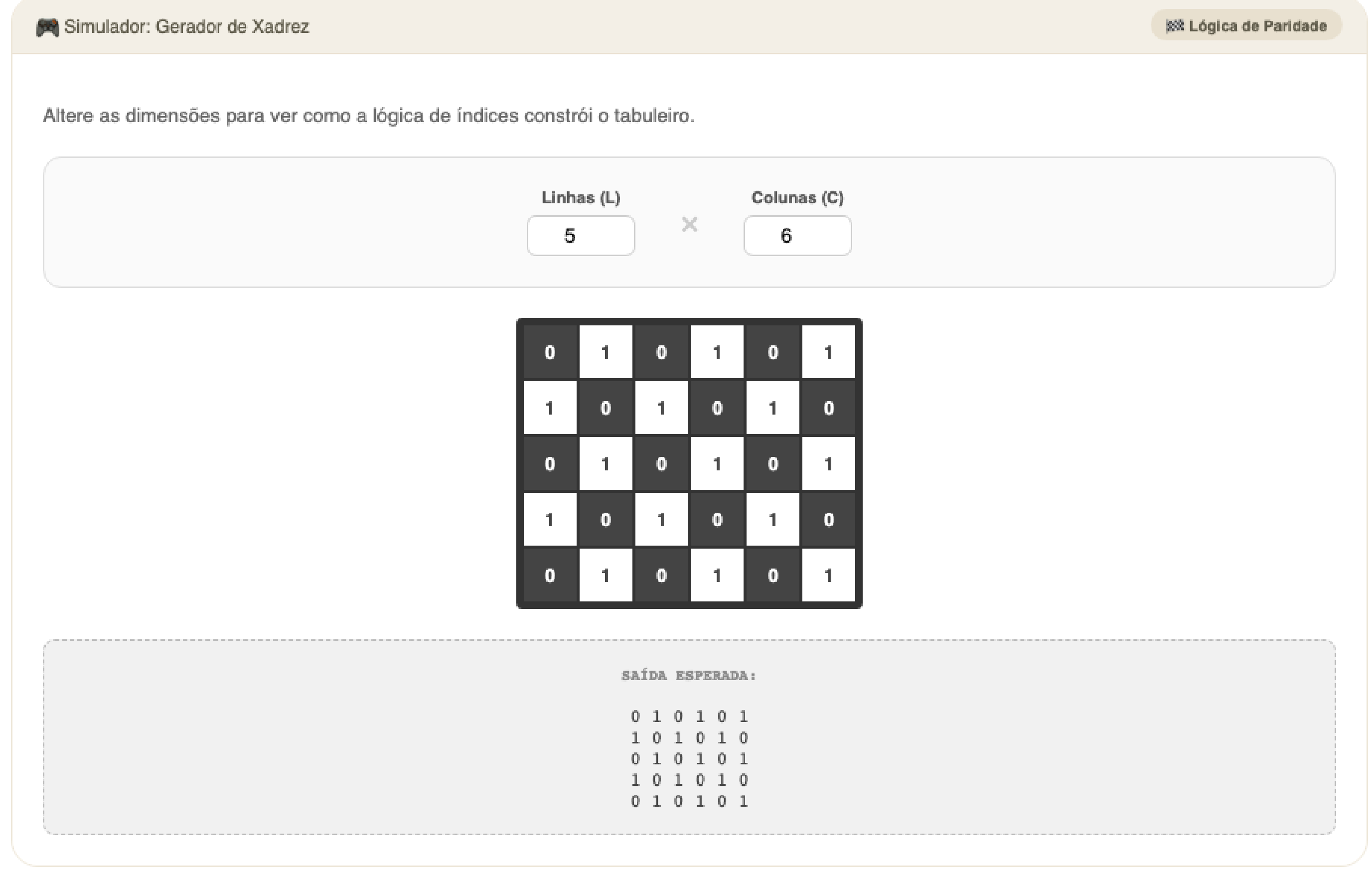


Figura 1.20: Simulador: Gerador de Xadrez

2. **Comentários:** Linhas opcionais que começam com `#` (devem ser ignoradas).
3. **Dimensões:** Dois inteiros representando **Largura** e **Altura**.
4. **Máximo:** Um inteiro representando a intensidade máxima (geralmente 255).

- Após o cabeçalho, seguem-se os dados dos pixels.

📌 **Importante**:

- **Leitura de Arquivo:** Você deve abrir o arquivo indicado no exemplo utilizando a função `open()` do Python.
- **Ordem de Saída:** Ao contrário da ordem presente no arquivo, a saída esperada deve estar no formato de tupla: `(Altura, Largura, Canais)`.
- Como arquivos PGM são em tons de cinza, o número de **Canais** é sempre **1**.
- Ver um simulador interativo para esta questão na **Figure 1.21** (ajuste as dimensões para ver como o cabeçalho ASCII é gerado).

1.16.10.1 🧠 Entendendo o formato PGM

O formato PGM é um dos mais simples para processamento de imagens. Por ser em texto puro, ele permite visualizar os metadados e até os valores dos pixels abrindo o arquivo em um bloco de notas:

Componente	Exemplo	Significado
Magic Number	`P2`	Identifica que é um PGM em formato de texto (ASCII).

Componente	Exemplo	Significado
Comentário	`# CREATOR...`	Linha informativa ignorada pelo processador.
Dimensões	`397 343`	**397** colunas (Largura) e **343** linhas (Altura).
Intensidade	`255`	Define o valor do branco puro (escala de 0 a 255).

1.16.10.2 📋 Tarefa (especificação para VPL)

Entrada:

Nenhuma entrada via teclado. O programa deve ler o arquivo `"aula01fig03b.pgm"` presente no diretório de execução.

Saída:

Uma tupla contendo `(Altura, Largura, 1)`.

1.16.10.3 📌 Exemplos

Nome do Arquivo	Saída Esperada	Observação
"aula01fig03b.pgm"	(343, 397, 1)	Note a inversão da ordem: Altura primeiro

ℹ Nota

O arquivo necessário para este EP será baixado automaticamente do repositório por meio do seguinte código:

```python
import os, urllib.request

BASE_URL = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/all/cap01/imagens"
file = "aula01fig03b.pgm"

opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
urllib.request.install_opener(opener)

for f in [file]:
    if not os.path.exists(f):
        url = f"{BASE_URL}/{f}"
        try:
            urllib.request.urlretrieve(url, f)
            print(f" Arquivo baixado: {f}")
        except urllib.error.HTTPError as e:
            print(f" Erro {e.code}: Não encontrado em {url}")
```

```
Arquivo baixado: aula01fig03b.pgm
```

Figura 1.21: Simulador: Estrutura do Arquivo PGM

```
%%writefile EP01_10.py
# sua solução
```

```
Writing EP01_10.py
```

```
TestSuite("EP01_10.py").run()
```

1.16.11 EP01_11 ↗ Análise de Vizinhança: Filtro de Máximo 1D

Nesta atividade, você deve implementar um filtro morfológico simples de máximo operando em um sinal unidimensional (vetor).

- Leia um inteiro **n**, representando o tamanho do vetor.
- Leia os **n** elementos inteiros que compõem o vetor original **v1**.
- Crie um novo vetor **v2**, onde cada posição i é o resultado da comparação entre o elemento atual e seus vizinhos imediatos:

$$v2[i] = \max(v1[i-1],\ v1[i],\ v1[i+1])$$

📌 **Importante**:

- **Bordas:** Nas extremidades do vetor (índices 0 e $n-1$), a vizinhança possui apenas dois elementos (o próprio e o único vizinho disponível). No índice 0, compare apenas $v1[0]$ e $v1[1]$. No último índice, compare apenas $v1[n-2]$ e $v1[n-1]$.

- **Saída:** Imprima o cabeçalho "v2:" seguido pelos valores do vetor resultante, um por linha.
- Ver um simulador interativo para esta questão na **Figure 1.22** (passe o mouse sobre os resultados para visualizar a janela de vizinhança utilizada no cálculo).

1.16.11.1 🧠 Por que analisar vizinhos?

Em processamento de imagens, o valor de um pixel raramente é isolado; ele depende do contexto ao seu redor. O **Filtro de Máximo** é a base da operação de **Dilatação** em morfologia matemática, servindo para:

Função	Efeito Visual
Realce	Expande estruturas brilhantes e "engorda" objetos claros.
Remoção de Ruído	Elimina pequenos pontos pretos (ruído de "sal e pimenta" escuro).
Preenchimento	Fecha pequenos buracos ou lacunas em formas binárias.

1.16.11.2 📋 Tarefa (especificação para VPL)

Entrada:

Um inteiro `n`.

Nas linhas seguintes, os `n` elementos inteiros do vetor.

Saída:

A string `v2:` na primeira linha.

Nas linhas seguintes, cada elemento de `v2` (um por linha).

1.16.11.3 📌 Exemplos

Entrada	Saída	Observação
5 10 20 5 30 15	v2: 20 20 30 30 30	No índice 1: max(10, 20, 5) = 20

```
%%writefile EP01_11.py
# sua solução
```

```
Writing EP01_11.py
```

```
TestSuite("EP01_11.py").run()
```

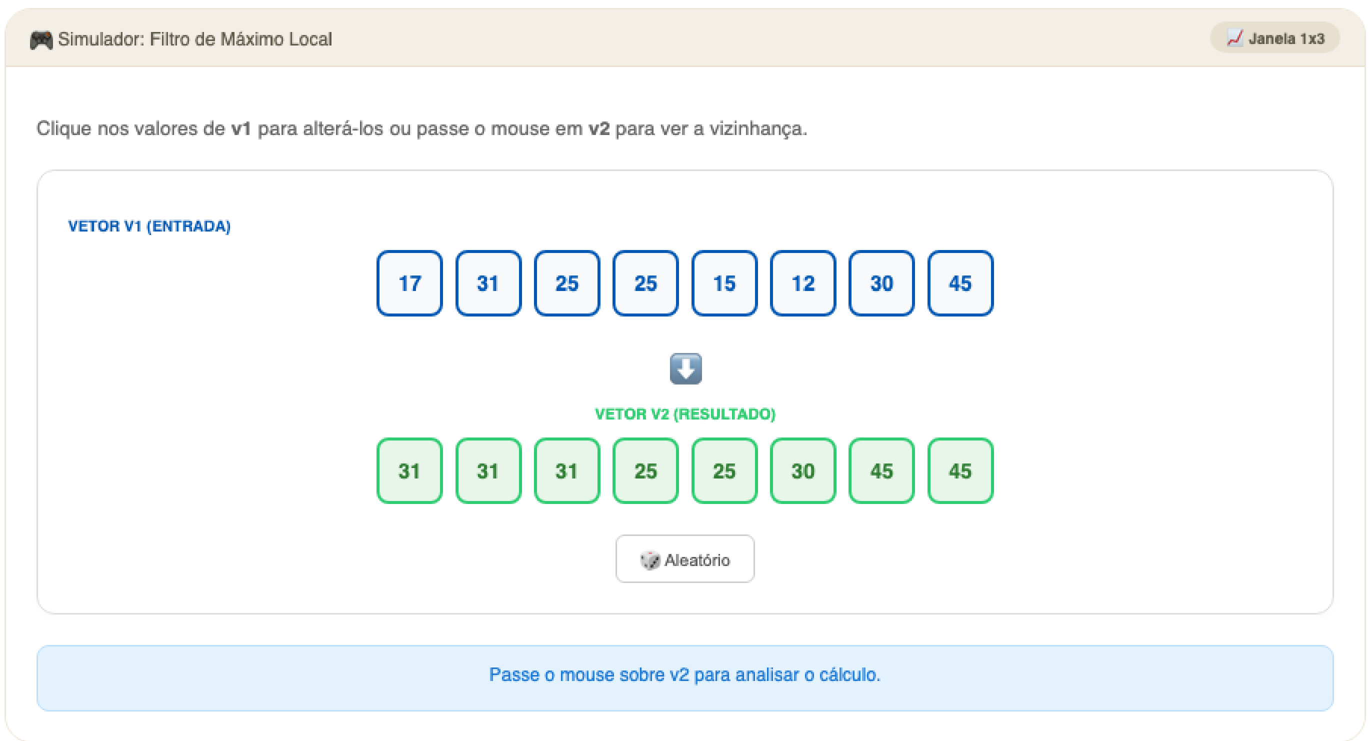


Figura 1.22: Simulador: Filtro de Máximo Local 1D

Capítulo 2

Da Captura ao Pixel - Amostragem, Quantização e Conectividade

Este capítulo aprofunda a compreensão da imagem digital, transitando da natureza física da captura para a sua representação matemática discreta. Investigamos como a luz se torna dado e como a organização espacial dos pixels define as relações de vizinhança e conectividade essenciais para algoritmos avançados de Visão Computacional.

2.1 Objetivos

Ao final deste capítulo, você será capaz de:

- Explicar o modelo físico de formação da imagem baseado em iluminação e refletância.
- Diferenciar os mecanismos da visão humana e dos sensores digitais.
- Compreender os processos de **amostragem** (discretização do espaço) e **quantização** (discretização da intensidade).
- Descrever as relações topológicas entre pixels: vizinhança, adjacência, conectividade e distâncias.
- Realizar transformações geométricas básicas (translação, rotação, escala) preservando a qualidade.
- Visualizar na prática os efeitos da variação da resolução espacial e da profundidade de bits.

2.2 O Olho e a Câmera - Elementos da Percepção Visual

A formação de uma imagem digital começa com a captura da luz refletida pelos objetos. Como ilustrado na Figure 2.1, tanto o olho humano quanto as câmeras digitais seguem princípios ópticos semelhantes para focar a luz em uma superfície sensível, embora utilizem mecanismos biológicos e eletrônicos distintos para a transdução do sinal.

2.2.1 Visão humana

O olho funciona como um sistema óptico complexo: a luz atravessa a córnea, o humor aquoso, a pupila (controlada pela íris) e o cristalino — que ajusta o foco dinamicamente — até atingir a retina. Na retina, encontram-se os fotorreceptores: os **cones** (6 milhões), concentrados na fóvea, são responsáveis pela visão de cores e detalhes, enquanto os **bastonetes** (120 milhões) garantem a visão em baixa luminosidade (visão escotópica), detectando apenas intensidades de cinza. O ponto cego é a região de onde parte o nervo óptico, carecendo de receptores.

2.2.2 Sensores digitais

Nas câmeras, os sensores de imagem desempenham o papel da retina. Os dois tipos mais comuns são o **CCD** (*Charge-Coupled Device* - Dispositivo de Carga Acoplada) e o **CMOS** (*Complementary Metal-Oxide-Semiconductor* - Semicondutor de Óxido Metálico Complementar). O sensor é composto por uma matriz de fotossítios (pixels) que acumulam carga elétrica proporcional à luz incidente.

Para a reconstrução de cores, utiliza-se o **Filtro de Bayer**, uma matriz de filtros coloridos que permite que cada pixel capture apenas uma componente de cor: vermelho, verde ou azul (RGGB - *Red, Green, Green, Blue*). Posteriormente, um **ADC** (*Analog-to-Digital Converter* - Conversor Analógico-Digital) quantiza essa carga em valores numéricos, definidos por uma profundidade de bits (ex.: 8 bits, resultando em 256 níveis de intensidade).

> **Curiosidade:** Embora o olho humano tenha milhões de receptores, a resolução de alta definição é restrita à fóvea (visão central), equivalente a aproximadamente 120 $\times$ 120 pixels. A percepção de uma cena completa em alta resolução é fruto de um intenso pós-processamento realizado pelo cérebro.

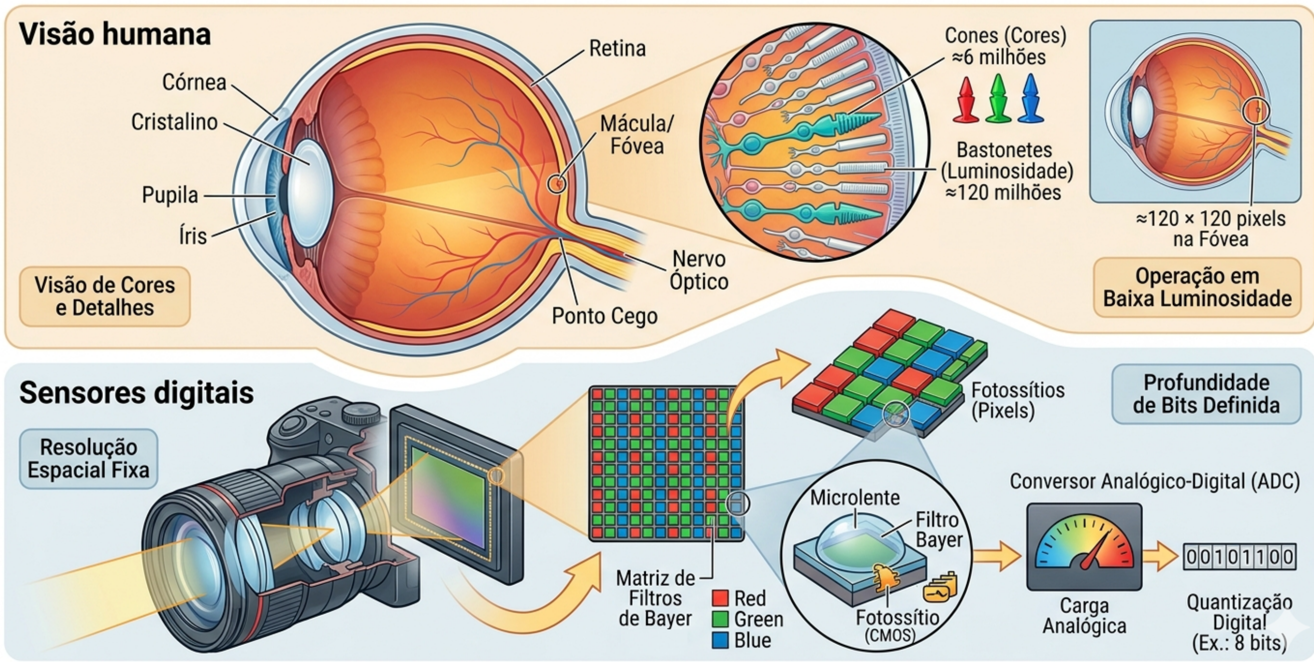


Figura 2.1: Comparativo didático entre o sistema visual biológico e o eletrônico: no topo, a anatomia do olho humano destacando a retina e os fotorreceptores (cones e bastonetes); abaixo, a estrutura de uma câmera digital detalhando o sensor CMOS, a matriz de filtros de Bayer (RGGB) e o processo de quantização digital realizado pelo ADC.

2.3 Ilusões de Ótica: Os Desafios da Percepção Visual

Enquanto os sensores digitais capturam a intensidade da luz de forma linear e objetiva, o sistema visual humano interpreta a cena com base em contexto, experiências prévias e mecanismos biológicos de sobrevivência. As ilusões de ótica não são "erros" do olho, mas evidências do intenso **pós-processamento cerebral** realizado no córtex visual.

2.3.1 Ambiguidade e Contexto

O cérebro busca constantemente dar sentido a padrões ambíguos. No exemplo do **Vaso de Rubin** (veja a Figure 2.2), a percepção alterna entre a figura (vaso) e o fundo (dois rostos), demonstrando que não conseguimos processar ambas as interpretações simultaneamente. Já a ilusão do **Elefante de Shepard** brinca com a nossa incapacidade de reconciliar linhas de contorno que sugerem volume em posições logicamente impossíveis.

2.3.2 Brilho e Contraste Local

Muitas ilusões decorrem da **inibição lateral**, mecanismo pelo qual neurônios vizinhos na retina competem entre si para realçar bordas. Na **Grade de Scintillating**, pontos escuros "fantasmas" parecem surgir nas interseções brancas devido a esse processamento local de contraste.

A ilusão da **Sombra no Tabuleiro de Adelson** é talvez a mais impactante para a Visão Computacional: o quadrado "A" e o quadrado "B" possuem exatamente o mesmo valor de cinza no sensor (ou arquivo digital), mas o cérebro "corrige" o brilho de "B" por entender que ele está sob uma sombra projetada, percebendo-o como mais claro.

2.3.3 Geometria e Perspectiva

A percepção de profundidade pode ser enganada por construções geométricas que desafiam a lógica tridimensional a partir de um ângulo de visão específico. A **Escada de Schröder** utiliza a ambiguidade da perspectiva para criar um objeto que parece subir ou descer dependendo de como é observado, evidenciando como a nossa interpretação de "cima" e "baixo" depende do ponto de fuga.

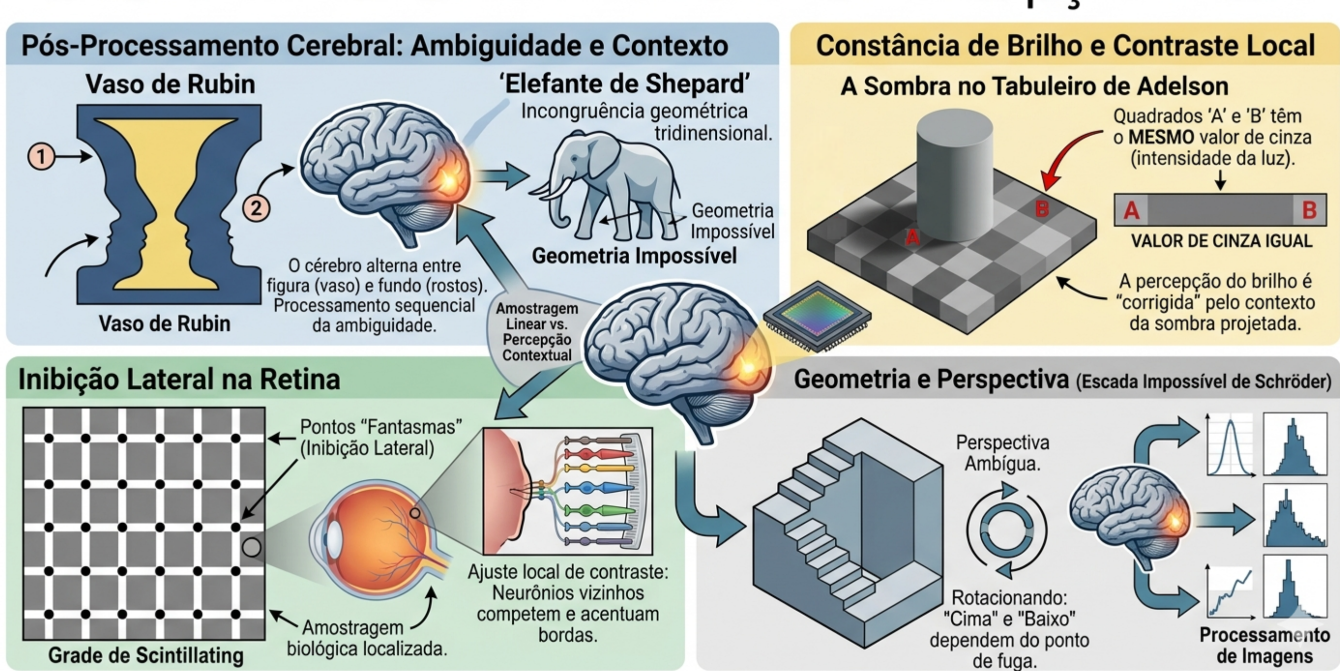


Figura 2.2: Coletânea de desafios perceptivos: (topo esquerdo) Vaso de Rubin — ambiguidade figura-fundo; (topo central) Elefante de Shepard — incongruência geométrica; (topo direita) Sombra de Adelson — constância de brilho baseada no contexto; (inferior esquerdo) Grade de pontos — inibição lateral; (inferior direito) Escada de Schröder.

2.4 O Modelo Matemático da Formação da Imagem

Uma imagem pode ser modelada como o produto de duas funções:

$$f(x, y) = i(x, y) \cdot r(x, y) \tag{2.1}$$

onde:

- $i(x, y)$ é a **iluminação** incidente sobre a cena (energia luminosa por unidade de área). Depende da fonte de luz.

- $r(x, y)$ é a **refletância** do objeto (fração da luz refletida). Depende das propriedades do material e da superfície.

Na prática, as duas componentes variam em faixas distintas: $i(x, y)$ varia lentamente no espaço, enquanto $r(x, y)$ pode variar rapidamente (texturas, bordas). O processamento de imagens frequentemente tenta separar ou compensar essas componentes (ex.: correção de iluminação não uniforme).

2.5 Digitalização: Amostragem e Quantização

Para transformar uma cena contínua em uma imagem digital, dois processos são necessários.

2.5.1 Amostragem - Discretização do espaço

A amostragem consiste em medir o valor da função $f(x, y)$ em pontos igualmente espaçados, formando uma matriz de M linhas (altura) e N colunas (largura). Cada elemento dessa matriz é um **pixel**. A **resolução espacial** é dada por $M \times N$. Quanto maior a resolução, mais detalhes espaciais são preservados, mas maior também o custo computacional e de armazenamento.

2.5.2 Quantização - Discretização da intensidade

A quantização associa a cada pixel um valor numérico discreto, geralmente representado por um inteiro de b bits. A **profundidade de bits** define o número de níveis de intensidade: 2^b. Imagens em tons de cinza costumam usar 8 bits (256 níveis). Imagens coloridas usam três canais de 8 bits (24 bits no total).

Ilustração: Se usarmos apenas 1 bit por pixel (preto e branco), perdemos todos os tons intermediários. Com 2 bits (4 níveis) já se percebe degradês grosseiros. Com 8 bits, o olho humano dificilmente percebe a discretização (visão contínua).

Erro de quantização

Erro de quantização é a diferença entre o valor analógico real e o valor discreto atribuído. Ele se manifesta como ruído de quantização, visível em regiões com gradiente suave quando se usa poucos bits (efeito de "posterização").

2.5.3 Laboratório prático: Efeitos da amostragem e quantização

Os experimentos a seguir mostram como a redução da resolução espacial (subamostragem) e da profundidade de bits degradam a qualidade visual. Use o código para explorar diferentes fatores e níveis de cinza.

```python
import os, importlib, urllib.request
import numpy as np
import matplotlib.pyplot as plt
import cv2

# Baixar morph.py se necessário
BASE_URL = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph"
for f in ["morph.py"]:
    if not os.path.exists(f):
        urllib.request.urlretrieve(f"{BASE_URL}/{f}", f)

import morph
importlib.reload(morph)
from morph import mm

print("  Ambiente pronto")
```

```
 Ambiente pronto
```

```
# Carregar imagem de exemplo
base = "https://upload.wikimedia.org/wikipedia/commons"

arquivo = (
    "Area_de_Prote%C3%A7%C3%A3o_Ambiental_"
    "Quilombos_do_M%C3%A9dio_Ribeira_-_Thomas-"
    "Fuhrmann_%282023-_02%29_Malacoptila_striata.jpg"
)

url = f"{base}/c/c5/{arquivo}"
caminho = "imagens/barbudo-rajado.jpg"

if not os.path.exists(caminho):
    os.makedirs("imagens", exist_ok=True)
    img_pil = mm.read(url, pil=True)   # retorna PIL Image com EXIF
    mm.write(img_pil, caminho)         # salva preservando EXIF
else:
    img_pil = mm.read(caminho, pil=True)

img_numpy = np.array(img_pil)

exif = img_pil._getexif()   # agora funciona - img_pil é PIL Image


img_color = mm.read(url)
img_gray0 = mm.gray(img_color)
print(f"Imagem original: {img_color.shape}")
print(f"Tipo da imagem: {type(img_color)}")
```

```
Imagem original: (3067, 2047, 3)
Tipo da imagem: <class 'numpy.ndarray'>
```

O experimento apresentado na Figure 2.3 ilustra o compromisso entre a **resolução espacial** e o **custo de armazenamento** em memória. O código utiliza a técnica de subamostragem por fatiamento (*slicing*) para reduzir a matriz original de pixels conforme um fator f, resultando em uma economia drástica de memória — por exemplo, um fator $f = 8$ reduz o tamanho do dado em 64 vezes (8^2). Para fins de comparação visual, as imagens reduzidas são restauradas às dimensões originais (512×512) através da interpolação por **vizinho mais próximo** (`nearest`). Este processo não recupera a informação perdida, mas torna evidente o efeito de **aliasing** e a estrutura de blocos (pixelização) gerada pela baixa densidade de dados da matriz amostrada.

```
def subsample_simple(image, f):
    # Subamostragem via fatiamento (slicing)
    reduced = image[::f, ::f]

    # Cálculo de memória em KB
    mem_kb = reduced.nbytes / 1024
    label = f"{reduced.shape[1]}x{reduced.shape[0]}, {int(mem_kb)} KB\n(Fator {f})"

    # Restaura o tamanho para visualização (H, W originais)
    res = mm.resize(reduced, (image.shape[1], image.shape[0]), method='nearest')
    return res, label

factors = [1, 4, 8, 12]
img_gray =img_gray0[820:1850, 890:1550] # Recorte para melhor visualização dos detalhes

# Gera os resultados e separa em listas para o mm.show
results = [subsample_simple(img_gray, f) for f in factors]
imgs_list = [r[0] for r in results]
titles_list = [r[1] for r in results]
```

```
mm.show(imgs_list, titles=titles_list, cols=4, figsize=(16, 12))
```

Figura 2.3: Efeito da subamostragem. Os títulos exibem as dimensões (W x H) e o tamanho da matriz em memória (KB).

O experimento na Figure 2.4 foca na **quantização de intensidade**, o processo de discretização da amplitude da função $f(x, y)$. Enquanto a subamostragem afeta a grade espacial, a redução da profundidade de bits limita a quantidade de tons de cinza disponíveis para representar o brilho.

Ao reduzir a profundidade de 8 bits (256 níveis) para valores menores, surge o efeito de **posterização**, onde os gradientes suaves de uma cena são substituídos por transições abruptas. No limite de 1 bit, a imagem torna-se estritamente binária, preservando apenas a silhueta e perdendo detalhes de textura e volume.

2.5.4 Análise Técnica

- **Domínio vs. Codomínio**: Note que a resolução espacial (dimensões da matriz) permanece constante em 512x512; o que se altera é apenas o codomínio da função da imagem.
- **Constância de Memória**: Observe nos títulos que o tamanho em KB não diminui. Isso ocorre porque o NumPy armazena cada pixel quantizado em um contêiner de 8 bits (`uint8`), independentemente de o valor real ser apenas 0 ou 1.
- **Percepção**: A degradação visual torna-se crítica abaixo de 4 bits, onde o olho humano começa a perceber as "fronteiras" artificiais criadas pela falta de tons intermediários.

```
def quantize_simple(image, bits):
    """Reduz a profundidade de bits e calcula metadados de memória."""
    levels = 2 ** bits
    # Normalização e quantização uniforme
    quantized = (np.floor(image / 256 * levels) / levels * 255).astype(np.uint8)

    # Cálculo de memória em KB
    mem_kb = quantized.nbytes / 1024
    label = f"{bits} bits ({levels} níveis)\n{int(mem_kb)} KB"

    return quantized, label

# Lista de bits para teste (8 é o padrão, 1 é o binário)
bits_test = [8, 4, 2, 1]

# Gera os resultados e separa em listas para o mm.show
```

```
results_q = [quantize_simple(img_gray, b) for b in bits_test]
imgs_q = [r[0] for r in results_q]
titles_q = [r[1] for r in results_q]

mm.show(imgs_q, titles=titles_q, cols=4, figsize=(16, 12))
```

Figura 2.4: Efeito da redução da profundidade de bits. Os títulos exibem a quantidade de bits, níveis e o tamanho em memória (KB).

```
# Exemplo de compactação real para economia de memória
import numpy as np

# Imagem binária em uint8 (512x512)
img_uint8 = np.zeros((512, 512), dtype=np.uint8)
print(f"Tamanho uint8: {img_uint8.nbytes / 1024} KB") # 256 KB

# Imagem compactada (Bit-packing)
img_packed = np.packbits(img_uint8)
print(f"Tamanho Compactado: {img_packed.nbytes / 1024} KB") # 32 KB
```

```
Tamanho uint8: 256.0 KB
Tamanho Compactado: 32.0 KB
```

A limitação de tipos de dados menores que um byte no ecossistema Python/NumPy deve-se à arquitetura de hardware, que endereça a memória em blocos de 8 bits (Bytes). Para que se mantenha a compatibilidade com o OpenCV e se garanta a eficiência, mapeiam-se até mesmo elementos binários para contêineres de 1 byte (`uint8` ou `bool8`).

Embora linguagens como ANSI C permitam a compactação de 8 pixels por byte (*bit-packing*), tal abordagem exige a descompactação constante para cálculos e impõe alta complexidade na manipulação de ponteiros. Conforme a Table 2.1, privilegia-se o uso de `uint8` pela facilidade no acesso a vizinhos e pela versatilidade em transformações geométricas. Além disso, métodos nativos do NumPy e OpenCV executam o processamento internamente em baixo nível (C/C++), o que torna operações vetorizadas mais rápidas do que implementações manuais com laços aninhados em Python.

Tabela 2.1: Comparativo entre estratégias de compactação e eficiência de processamento.

Característica	Python (NumPy/OpenCV)	ANSI C (Bit-packing)
Menor Unidade	1 Byte (8 bits)	1 Bit
Memória (Binária)	256 KB (para 512x512)	32 KB (para 512x512)
Velocidade	Alta (Vetorização em C)	Variável (Lenta se houver bit-shift)
Complexidade	Baixa: Métodos prontos	Alta: Ponteiros e Máscaras

2.6 Relações entre Pixels - Topologia da Imagem

Os pixels não são elementos isolados; suas posições relativas definem importantes conceitos para processamento.

2.6.1 Vizinhança

Dado um pixel de coordenadas (x, y), definem-se dois tipos principais de vizinhança (para imagens em *grid* retangular):

- **Vizinhança-4** (von Neumann): inclui os pixels nas posições $(x-1, y)$, $(x+1, y)$, $(x, y-1)$, $(x, y+1)$.
- **Vizinhança-8** (Moore): inclui todos os oito pixels adjacentes (acrescenta os quatro diagonais).

A escolha da vizinhança influencia operações como detecção de bordas, cálculo de gradientes e conectividade.

```
# Criação de uma matriz 3x3 para exemplo topológico
viz = np.zeros((3, 3), dtype='uint8')

# Pixel central (p)
viz[1, 1] = 150

# Definindo Vizinhança-4 (N4) com valor diferente para destaque
viz[0, 1] = viz[2, 1] = viz[1, 0] = viz[1, 2] = 255

# Exibição da matriz para análise de coordenadas
mm.drawImagePlt(viz, scale=40)
```

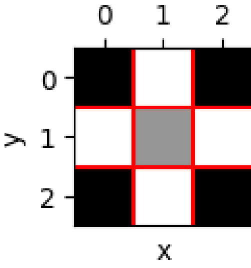


Figura 2.5: Ilustração de vizinhanças 4 em uma matriz 3x3. No centro (1,1), o pixel de interesse.

2.6.2 Adjacência, conectividade e caminhos

Dois pixels são **adjacentes** se estão em contato segundo uma vizinhança definida **e** satisfazem um critério de valor (ex.: mesmo nível de intensidade). Uma **conectividade** define uma relação de equivalência entre pixels que formam uma região conexa. Um **caminho** é uma sequência de pixels adjacentes.

A conectividade-4 e conectividade-8 podem produzir resultados diferentes na segmentação e no cálculo de componentes conexas (etiquetagem). Por exemplo, um padrão de tabuleiro de xadrez pode ser completamente desconexo em 4-vizinhança, mas totalmente conexo em 8-vizinhança.

2.6.3 Distâncias entre pixels

As métricas de distância são fundamentais para quantificar a proximidade física e a conectividade entre os elementos que compõem a grade digital. Conforme demonstrado na Table 2.2, a escolha da métrica define o custo de deslocamento entre pixels e altera o comportamento de algoritmos de segmentação e análise morfológica.

Para medir a distância entre dois pixels $p(x_1, y_1)$ e $q(x_2, y_2)$, utilizam-se diferentes funções métricas que impõem restrições de movimento distintas sobre o *grid*:

Tabela 2.2: Comparativo de métricas de distância aplicadas à malha de pixels.

Métrica	Definição	Interpretação
Euclidiana	$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$	Distância em linha reta (contínua)
Manhattan **(*City block*)**	$\|x_1 - x_2\| + \|y_1 - y_2\|$	Movimentos horizontais + verticais
Chebyshev **(Tabuleiro)**	$\max(\|x_1 - x_2\|, \|y_1 - y_2\|)$	Movimentos incluindo diagonais

Essas distâncias são aplicadas em diversos contextos de PDI, incluindo algoritmos de interpolação geométrica, transformadas de distância, crescimento de regiões e análise de formas.

> **Exemplo prático**
>
> Considerando dois pixels com deslocamentos relativos $\Delta x = 3$ e $\Delta y = 4$:
> - **Euclidiana**: $\sqrt{3^2 + 4^2} = 5$ (hipotenusa do triângulo retângulo).
> - ***Manhattan***: $3 + 4 = 7$ (soma dos catetos).
> - ***Chebyshev***: $\max(3, 4) = 4$ (predomínio do maior deslocamento).

2.7 Armazenamento de Imagens

A escolha do formato de arquivo é um passo decisivo no fluxo de processamento, pois determina como os dados de amostragem e quantização serão preservados ou descartados. Conforme se apresenta na Table 2.3, cada extensão equilibra de forma distinta a fidelidade dos dados e a eficiência de armazenamento.

As imagens digitais podem ser armazenadas em diversos formatos, cada um com características que impactam o processamento posterior:

Tabela 2.3: Principais formatos de armazenamento e suas aplicações em PDI.

Formato	Características	Uso típico
PGM	Formato simples de mapa de cinzas (texto ou binário).	Pesquisa acadêmica e ferramentas Unix.
BMP	Não comprimido (ou compressão simples).	Windows, aplicações legado.
PNG	Compressão sem perdas (*lossless*).	Web, imagens com transparência.
JPEG	Compressão com perdas (*lossy*), ideal para fotografias.	Fotos, imagens médicas (uso moderado).
TIFF	Suporta múltiplas camadas e compressão variada.	Editoração, arquivamento.
RAW	Dados brutos (sensor ou matriz sem cabeçalho).	Fotografia profissional e exames médicos.

Os metadados de uma imagem incluem parâmetros como largura, altura, profundidade de bits e codificação de cor. Em formatos científicos, preservam-se também informações de calibração e detalhes da captura. Ao utilizar-se a função `mm.read()`, a biblioteca `morph` preserva automaticamente esses dados para que se respeitem as propriedades originais da imagem.

Em contextos científicos e hospitalares, privilegia-se o padrão **DICOM** para garantir que não haja perda de precisão diagnóstica. Repositórios públicos como o **The Cancer Imaging Archive (TCIA)**, o **Alzheimer's Disease Neuroimaging Initiative (ADNI)** e o **PhysioNet** disponibilizam vastos conjuntos de dados neste formato, incluindo metadados clínicos anonimizados que são essenciais para a investigação científica.

2.7.1 Exemplo: Extração de Metadados e Localização GPS

Diferente da matriz de pixels pura obtida pela leitura convencional — como na imagem do pássaro apresentada no início deste capítulo —, o uso do argumento `info=True` no método `mm.read()` altera a natureza do objeto retornado (ver Figure 2.6). Enquanto o `imread` do OpenCV retorna um `numpy.ndarray` (padrão utilizado quando `info=False`), a leitura com informações preservadas retorna um objeto especializado da biblioteca Pillow, capaz de interpretar o cabeçalho **EXIF**.

O cabeçalho **EXIF** (*Exchangeable Image File Format*) funciona como um repositório técnico da captura, permitindo que o objeto Pillow interprete uma vasta gama de informações que vão muito além das coordenadas de GPS. Ao utilizar `info=True`, o sistema ganha acesso ao "DNA" da imagem, incluindo metadados de hardware (marca e modelo da câmera), configurações ópticas (abertura, distância focal e tempo de exposição) e parâmetros de iluminação (uso de flash e balanço de branco).

Essa distinção é importante no PDI, pois transforma a matriz de amostragem em um conjunto de dados contextualizado, onde as características físicas do sensor e da lente podem ser utilizadas para normalizar brilhos ou corrigir distorções geométricas.

```python
from PIL.ExifTags import TAGS
import os, numpy as np

base    = "https://upload.wikimedia.org/wikipedia/commons"
arquivo = (
    "Area_de_Prote%C3%A7%C3%A3o_Ambiental_"
    "Quilombos_do_M%C3%A9dio_Ribeira_-_Thomas-"
    "Fuhrmann_%282023-_02%29_Malacoptila_striata.jpg"
)
url     = f"{base}/c/c5/{arquivo}"
caminho = "imagens/barbudo-rajado.jpg"

# 1. Leitura - preserva objeto PIL com EXIF
if not os.path.exists(caminho):
    os.makedirs("imagens", exist_ok=True)
    img_obj = mm.read(url, pil=True)
    mm.write(img_obj, caminho)      # salva preservando EXIF
else:
    img_obj = mm.read(caminho, pil=True)

img_numpy = np.array(img_obj)      # conversão para NumPy

# 2. Extração e conversão de GPS (Tag 34853)
exif = img_obj._getexif()
if exif and (gps := exif.get(34853)):
    to_dec = lambda dms, ref: float(-(dms[0]+dms[1]/60+dms[2]/3600) if ref in 'SW'
                                   else (dms[0]+dms[1]/60+dms[2]/3600))
    lat, lon = to_dec(gps[2], gps[1]), to_dec(gps[4], gps[3])
    print(f"GPS Decimal: {lat:.6f}, {lon:.6f}")
    print(f"Maps: https://www.google.com/maps/search/?api=1&query={lat},{lon}")
```

```
# 3. Diagnóstico de tipos, dimensões e acesso a pixels
print(f"\nTipo PIL  : {type(img_obj)}   | Dimensões (x,y): {img_obj.size}")
print(f"Pillow (0,0): {img_obj.getpixel((0, 0))}")
print(f"Tipo NumPy: {type(img_numpy)} | Dimensões [y,x,c]: {img_numpy.shape}")
print(f"NumPy [0,0]: {img_numpy[0, 0]}")

# 4. Exibição
mm.show(img_numpy, scale=30)
```

```
GPS Decimal: -24.587955, -48.629758
Maps: https://www.google.com/maps/search/?api=1&query=-24.587955,-48.629758333333335

Tipo PIL  : <class 'PIL.JpegImagePlugin.JpegImageFile'>   | Dimensões (x,y): (2047, 3067)
Pillow (0,0): (58, 96, 0)
Tipo NumPy: <class 'numpy.ndarray'> | Dimensões [y,x,c]: (3067, 2047, 3)
NumPy [0,0]: [58 96  0]
```

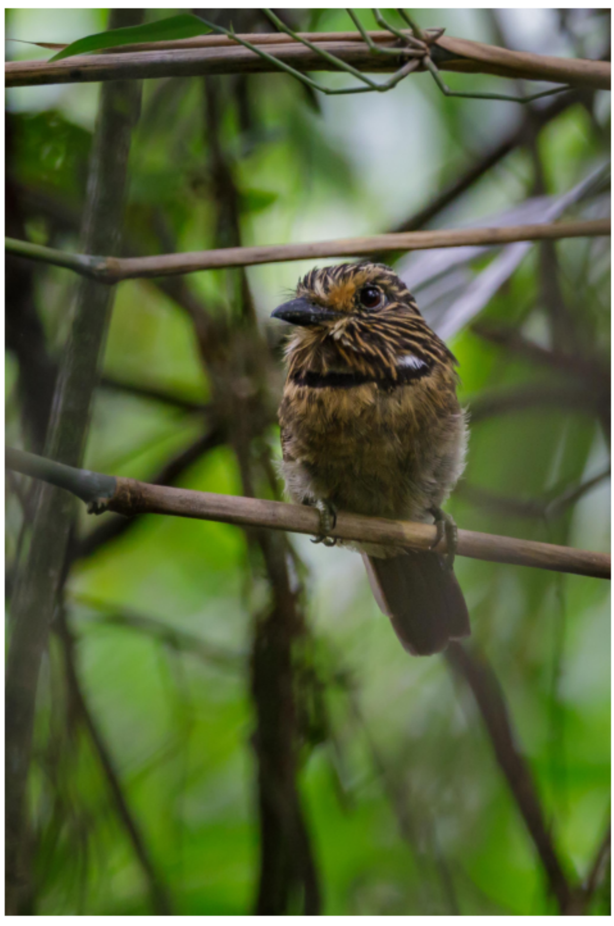


Figura 2.6: APA Quilombos do Médio Ribeira - Barbudo-rajado (Malacoptila striata). Crédito: Thomas Fuhrmann (CC BY-SA 4.0).

ℹ Nota Pedagógica: A sutil diferença das dimensões

Repare que a representação das dimensões muda conforme a estrutura de dados utilizada:

- **No Pillow (`.size`):** Retorna `(Largura, Altura)`. É uma visão orientada ao arquivo de imagem.
- **No NumPy (`.shape`):** Segue a convenção matemática de matrizes: `(Linhas/Altura, Colunas/Largura, Canais)`.

Esta distinção é fundamental para evitar erros de indexação ao implementar filtros manuais. Enquanto

o objeto Pillow carrega o **"onde"** e o **"quando"** (contexto), o array NumPy carrega o **"quanto"** de luz (intensidade) existe em cada ponto da imagem.

2.7.2 Por que esta separação é importante?

Ao carregar uma imagem via `cv2.imread()`, o resultado é um `<class 'numpy.ndarray'>`, que contém estritamente os valores numéricos resultantes da **quantização** e **amostragem**. No entanto, ao utilizar `info=True`, o `mm.read()` retorna um objeto da classe `PIL.JpegImagePlugin.JpegImageFile`.

Esta classe mantém o arquivo "aberto" para permitir o acesso ao contexto da captura antes que os dados sejam convertidos em uma matriz bruta. Essa separação é vital: pixels servem para algoritmos; metadados servem para georreferenciamento, catalogação científica e correções baseadas no hardware de aquisição.

2.8 Transformações Geométricas Básicas

As transformações geométricas alteram a posição dos pixels, mantendo os valores de intensidade. São fundamentais para alinhamento, correção de distorções e aumento de dados (*data augmentation*) em aprendizado de máquina.

Uma **transformação afim** é qualquer mapeamento que preserve colinearidade (pontos sobre uma reta permanecem sobre uma reta) e razões de distâncias entre pontos colineares. Em 2D, toda transformação afim pode ser expressa em **coordenadas homogêneas** por uma matriz 3×3:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{T}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.2}$$

A sub-matriz 2×2 superior esquerda $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ controla rotação, escala e cisalhamento; o vetor $(t_x, t_y)^\top$ controla a translação. As transformações geométricas mais usadas em PDI — translação, rotação e escala — são casos particulares de $\mathbf{T}$, e podem ser **compostas** por multiplicação de matrizes, na ordem $\mathbf{T} = \mathbf{T}_n \cdots \mathbf{T}_2 \mathbf{T}_1$.

i Transformação inversa e interpolação

Na implementação prática (`cv2.warpAffine`), aplica-se a **transformação inversa**: para cada pixel (x', y') da imagem destino, calcula-se a posição de origem $(x, y) = \mathbf{T}^{-1}(x', y')$ e interpola-se o valor. Isso evita buracos na imagem resultante causados pelo mapeamento direto de pixels inteiros para posições não inteiras.

2.8.1 Translação

A translação é a transformação afim mais simples: desloca todos os pixels por um vetor (t_x, t_y). Em coordenadas homogêneas, é expressa pela matriz:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \Rightarrow \quad \begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases} \tag{2.3}$$

A terceira linha da matriz garante que a operação permaneça no espaço afim, permitindo que translação, rotação e escala sejam combinadas por simples multiplicação de matrizes. Na prática, `cv2.warpAffine` usa apenas as duas primeiras linhas (matriz 2×3), pois a terceira é sempre $[0, 0, 1]$.

Pixels deslocados para além da área original são descartados; áreas descobertas são preenchidas com 0 (preto). Ver um exemplo na Figure 2.7.

```
img_tx1 = mm.translate(img_gray, 50, 50)
img_tx2 = mm.translate(img_gray, 100, 50)
mm.show([img_gray, img_tx1, img_tx2],
        titles=["Original", "Translação (50,50)", "Translação (100,50)"],
        cols=3, figsize=(16, 12))
```

Figura 2.7: Exemplo de translação da imagem do pássaro com deslocamentos (50,50) e (100,50).

2.8.2 Rotação

A rotação por ângulo θ em torno de um ponto central (c_x, c_y) é composta por três transformações afins: translação para a origem, rotação pura e translação de volta. A matriz resultante é:

$$\mathbf{T}_{\text{rot}} = \begin{bmatrix} \cos\theta & -\sin\theta & c_x(1-\cos\theta) + c_y \sin\theta \\ \sin\theta & \cos\theta & c_y(1-\cos\theta) - c_x \sin\theta \\ 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

Na implementação, `cv2.getRotationMatrix2D` gera diretamente as duas primeiras linhas de $\mathbf{T}_{\text{rot}}$ (matriz 2×3 para `warpAffine`), aceitando também um fator de escala s que multiplica $\cos\theta$ e $\sin\theta$. Ver exemplo na Figure 2.8.

```
img_rot30 = mm.rotate(img_gray, 30, interp='bilinear')
img_rot45 = mm.rotate(img_gray, 45, interp='bilinear')

mm.show([img_gray, img_rot30, img_rot45],
        titles=["Original", "Rotação 30°", "Rotação 45°"],
        cols=3, figsize=(16, 12))
```

Figura 2.8: Exemplo de rotação da imagem do pássaro em 30° e 45° usando interpolação bilinear.

2.8.3 Escala (Redimensionamento)

O redimensionamento por fatores (s_x, s_y) é uma transformação afim com matriz:

$$\mathbf{T}_{\text{escala}} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

Quando $s > 1$ (ampliação), pixels da imagem destino mapeiam para posições não inteiras na origem — exigindo **interpolação** para estimar o valor. Quando $s < 1$ (redução), múltiplos pixels de origem contribuem para um único pixel destino — exigindo **decimação**. Os três métodos disponíveis em `mm.resize` diferem na qualidade e no custo computacional, conforme a Table 2.4:

Tabela 2.4: Métodos de interpolação disponíveis em `mm.resize` e seus respectivos números de vizinhos utilizados no cálculo.

Método	Vizinhos usados	Característica
`'nearest'`	1	Rápido; produz efeito de blocos (*pixelation*)
`'bilinear'`	4	Bom compromisso qualidade/custo; bordas suaves
`'bicubic'`	16	Maior nitidez; preferido em softwares profissionais

```
# 1. Recorte da região do bico
y, x, offset = 210, 40, 40
crop = img_gray[y-offset:y+offset, x-offset:x+offset]
print(f"Imagem: {img_gray.shape} | Crop: {crop.shape}")

# 2. Ampliar 4× com mm.resize
crop_nearest  = mm.resize(crop, 4, method='nearest')
crop_bilinear = mm.resize(crop, 4, method='bilinear')

# 3. Exibição comparativa
```

```
mm.show(
    [crop, crop_nearest, crop_bilinear],
    titles=["Original (recorte)", "Vizinho mais próximo (4×)", "Bilinear (4×)"],
    cols=3, figsize=(16, 12), dpi=200
)
```

```
Imagem: (1030, 660) | Crop: (80, 80)
```

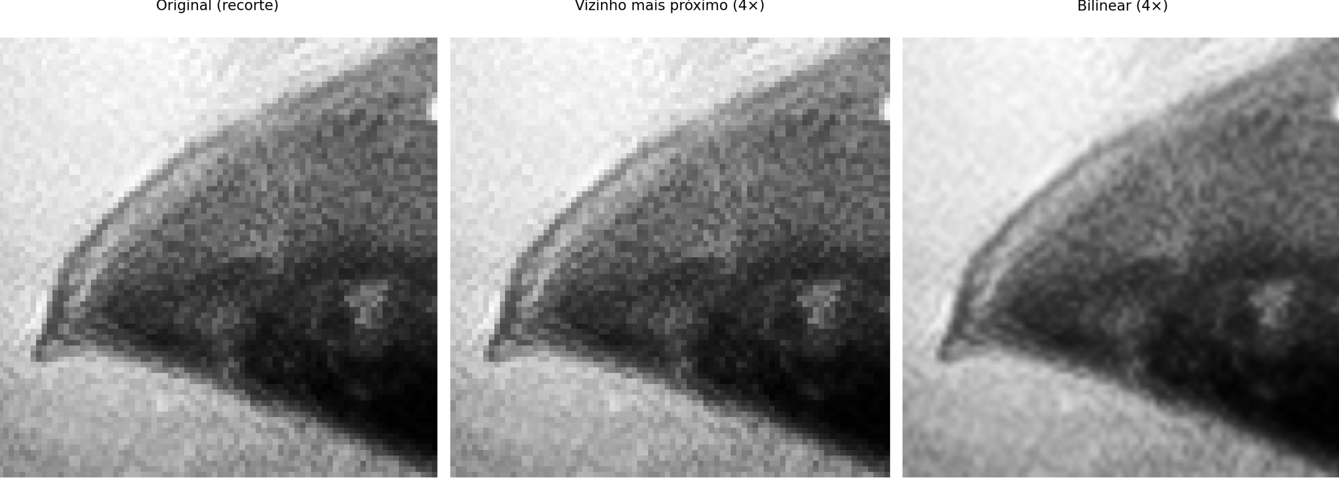


Figura 2.9: Comparação de interpolação com zoom no detalhe do olho (recorte 60×60, ampliado 4×). Note o efeito de blocos no vizinho mais próximo vs. a suavização na bilinear.

2.8.4 Cisalhamento (*Shear*)

O cisalhamento é uma transformação afim que distorce a imagem ao deslocar cada pixel proporcionalmente à sua posição em um eixo. A matriz geral combina cisalhamento horizontal (sh_x) e vertical (sh_y):

$$\mathbf{T}_{\text{cisalh}} = \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

Para $sh_x \neq 0$ e $sh_y = 0$, cada linha é deslocada horizontalmente proporcionalmente à sua posição vertical — produzindo o efeito de “inclinação” característico. Ver exemplo na Figure 2.10.

```
img_shx = mm.shear(img_gray, shx=0.3)
img_shy = mm.shear(img_gray, shy=0.3)
img_shc = mm.shear(img_gray, shx=0.2, shy=0.2)

mm.show(
    [img_gray, img_shx, img_shy, img_shc],
    titles=["Original", "Horiz. (shx=0.3)", "Vert. (shy=0.3)", "Combinado (0.2, 0.2)"],
    cols=4, figsize=(16, 12)
)
```

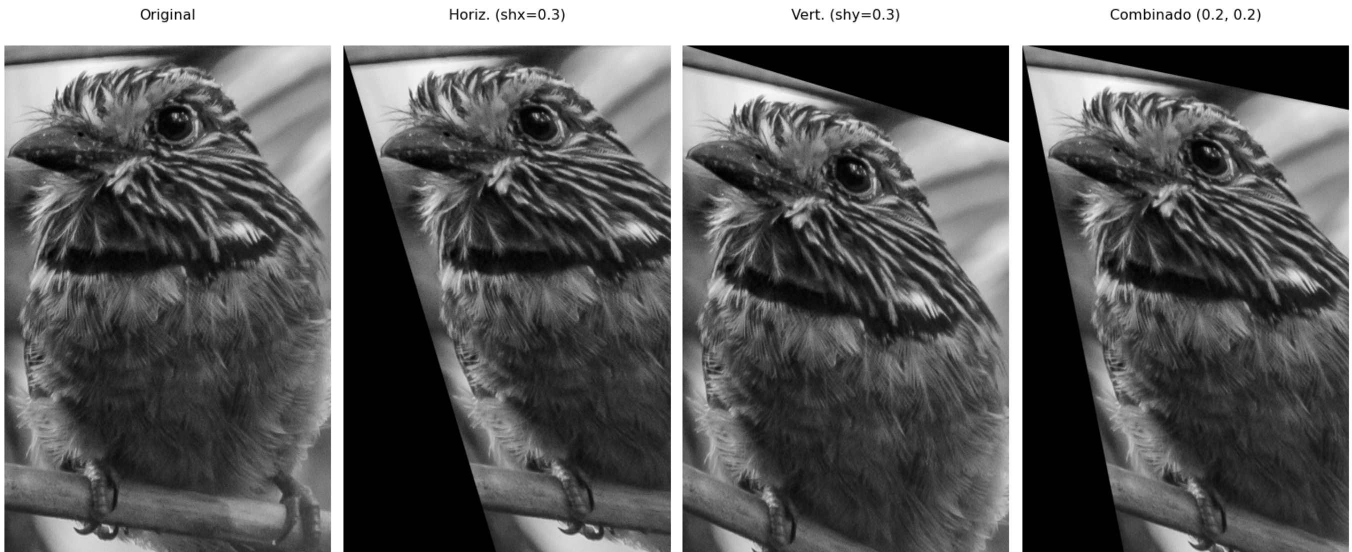


Figura 2.10: Exemplo de cisalhamento da imagem do pássaro: horizontal (shx=0.3), vertical (shy=0.3) e combinado (shx=0.2, shy=0.2).

2.9 Resumo

Neste capítulo foram apresentados os fundamentos de digitalização e topologia de imagens:

- **Formação da imagem:** $f(x, y) = i(x, y) \cdot r(x, y)$.
- **Amostragem:** discretização do espaço → resolução espacial $M \times N$.
- **Quantização:** discretização da intensidade → profundidade de bits b.
- **Relações topológicas:** vizinhança-4, vizinhança-8, conectividade, distâncias (Euclidiana, Manhattan, Chebyshev).
- **Transformações geométricas:** translação, rotação, escala (com interpolação bilinear ou vizinho mais próximo).
- **Formatos de arquivo:** BMP, PNG, JPEG, TIFF, RAW; cada um com diferentes compromissos entre qualidade e tamanho.

O Capítulo 3 abordará **operações espaciais** como convolução, filtragem e morfologia matemática (erosão, dilatação).

2.10 🤖 Uso do NotebookLM como Tutor Complementar

Nesta edição, incentivamos o uso do **NotebookLM** como ferramenta complementar de aprendizagem. Essa ferramenta de IA utiliza exclusivamente os documentos fornecidos pelo autor como base de conhecimento, garantindo respostas coerentes com o conteúdo do livro.

Para cada capítulo, preparamos um projeto específico na plataforma. Para uma experiência de estudo ampliada, utilize o acesso abaixo:

🎓 Estude com o Tutor Inteligente

Para interagir com o conteúdo deste capítulo, acesse o link a seguir. O ambiente contém materiais didáticos em diferentes formatos, gerados a partir do **PDF** do capítulo. Na plataforma, explore especialmente as opções **Guia de Estudo** e **Conversa** para aprofundar sua compreensão.
🚀 ACESSAR NOTEBOOKLM: CAPÍTULO 02

2.11 Lista de Exercícios

1. **(15%)** Explique, com suas próprias palavras, a diferença entre **amostragem** e **quantização**. Dê um exemplo concreto de cada uma no contexto de uma imagem digital.
2. **(15%)** Considere uma imagem com resolução espacial de 1024×768 pixels e profundidade de 24 bits (8 bits por canal RGB). Calcule o tamanho total não comprimido da imagem em bytes e em megabytes.
3. **(20%)** Utilizando o código do laboratório, modifique o fator de subamostragem para 3 e para 6. Descreva visualmente o que ocorre com as bordas dos objetos. O que é o efeito de *aliasing*?
4. **(20%)** Para a imagem em tons de cinza, aplique quantização com 3 bits (8 níveis) e 5 bits (32 níveis). Compare os resultados e explique por que 5 bits já pode ser considerado suficiente para muitas aplicações.
5. **(15%)** Dados dois pixels $A = (10, 20)$ e $B = (15, 25)$, calcule as distâncias Euclidiana, Manhattan e Chebyshev entre eles.
6. **(15%)** Usando a função `mm.rotate`, gire a imagem do pássaro em ângulos de 90°, 180° e 270° com interpolação bilinear. Compare com a rotação usando `method='nearest'`. Em quais situações a interpolação vizinho mais próximo ainda é útil?

Referências do Capítulo

A fundamentação teórica deste capítulo baseia-se nas seguintes obras:

- Gonzalez; Woods (2018) para os conceitos de amostragem, quantização e relações entre pixels.
- Szeliski (2022) para transformações geométricas e conectividade.
- Bradski; Kaehler (2008) para a implementação prática com OpenCV e `morph.py`.

2.12 💻 Parte Prática com Exercícios de Programação

🎯 Objetivo deste Caderno

O caderno permite desenvolver, validar, organizar e testar soluções de **Exercícios de Programação (EPs)** em ambientes interativos, como o Colab, com os mesmos casos de teste do Moodle, copiando para lá apenas na hora de registrar a nota oficial.

Download

Baixe `morph.py` e `testsuite.py` executando a célula abaixo:

```python
import os, sys, importlib, inspect, urllib.request

# URLs do repositório
BASE_URL = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph"
for f in ["morph.py", "testsuite.py"]:
    if not os.path.exists(f):
        urllib.request.urlretrieve(f"{BASE_URL}/{f}", f)

import morph, testsuite
importlib.reload(morph); importlib.reload(testsuite)
from morph import mm
```

```
from testsuite import TestSuite

print(f" Ambiente pronto. Morph: {morph.__version__} | TestSuite: {testsuite.__version__}")
```

```
 Ambiente pronto. Morph: 1.1.0 | TestSuite: 1.1.0
```

2.12.1 EP02_01 ⊙ Ajuste de Brilho e Contraste

Nesta atividade, você deve implementar uma transformação linear de intensidade para ajustar o brilho e o contraste de uma imagem.

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz.
- Leia um valor real α (contraste) e um inteiro β (brilho).
- Leia os valores inteiros da matriz.
- Para cada pixel p, calcule o novo valor p' usando a fórmula:

$$p' = \text{clip}(\text{round}(\alpha \times p + \beta))$$

- Imprima a matriz resultante com os valores finais.

📌 **Importante**:

- **Saturação (Clip):** O resultado final deve ser obrigatoriamente um inteiro no intervalo $[0, 255]$. Valores menores que 0 tornam-se 0, e valores maiores que 255 tornam-se 255.
- **Arredondamento:** Realize o arredondamento matemático antes de converter para inteiro e aplicar o clip.
- Ver um simulador interativo para esta questão na Figure 2.11 (manipule os parâmetros α e β para visualizar o efeito de realce e correção).

2.12.1.1 🧠 Transformação Linear de Intensidade

Ajustar o brilho e o contraste são operações fundamentais em PDI. Elas permitem corrigir imagens subexpostas (escuras demais) ou melhorar a visibilidade de detalhes sutis:

Parâmetro	Função	Efeito
α **(Alpha)**	Multiplicador	Ajusta o **Contraste**. > 1 aumenta a diferença entre tons; < 1 diminui.
β **(Beta)**	Aditivo	Ajusta o **Brilho**. Valores positivos clareiam; negativos escurecem.
Clip	Limitador	Garante que o pixel permaneça dentro do padrão de 8 bits $(0 - 255)$.

2.12.1.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

A terceira linha contém `alpha` (α) e `beta` (β).

As linhas seguintes contêm os elementos da matriz.

Saída:

A matriz transformada com `L` linhas e `C` colunas.

2.12.1.3 📌 Exemplos

Entrada	Saída	Observação
1 4 1.5 -30 0 100 180 255	0 120 240 255	Note o efeito de saturação no último pixel

---

Executando os Testes

Para rodar os testes, execute `TestSuite("EP01_01.extensão").run()` numa nova célula, trocando a extensão pela da linguagem usada (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). O sistema baixa os casos de teste do GitHub, executa o programa e calcula a nota automaticamente.

Exemplo de teste de sqrt em Python, com `timeit` isolando cada operação:

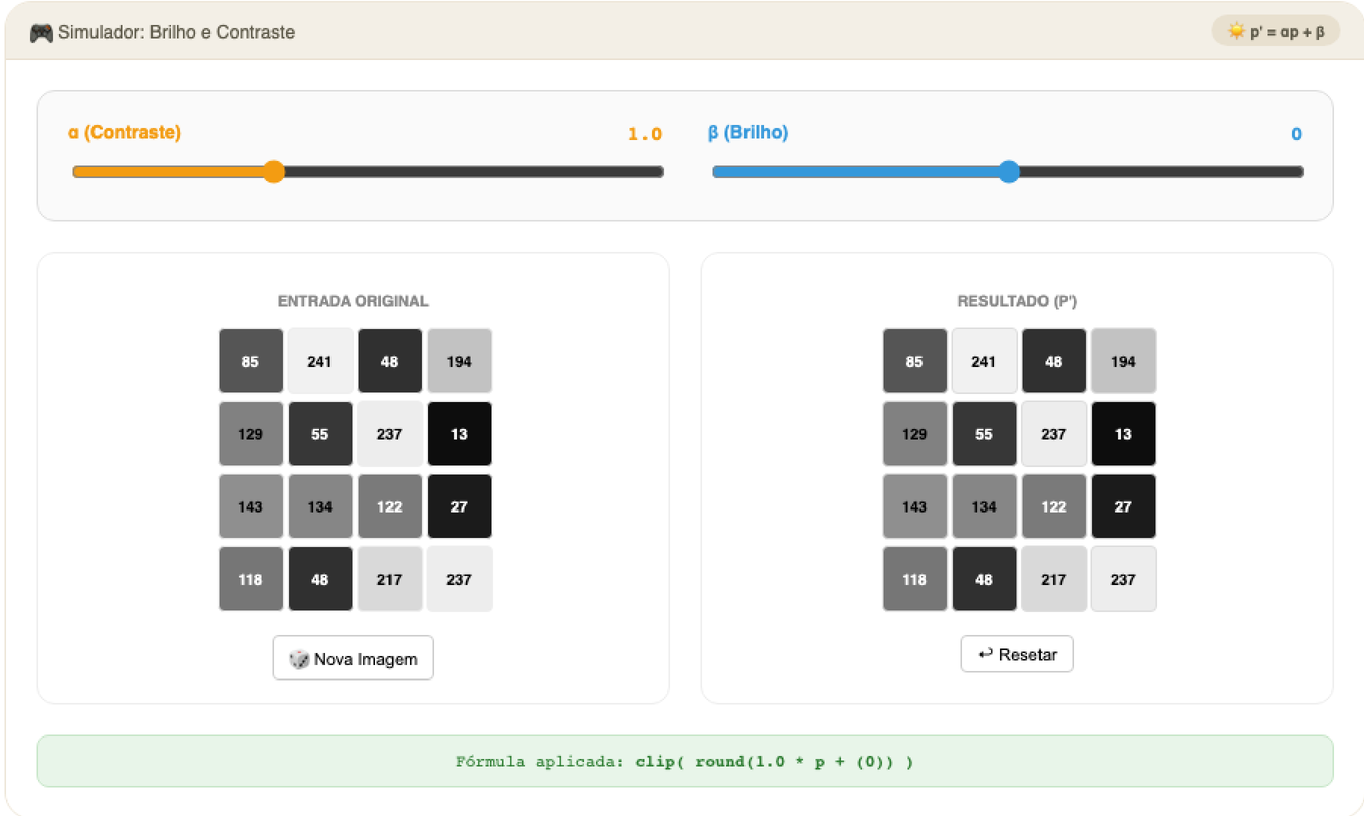


Figura 2.11: Simulador: Ajuste de Brilho e Contraste

```
%%writefile EP02_01.py
# Código Python
```

```
Writing EP02_01.py
```

```
TestSuite("EP02_01.py").run()
```

2.12.2 EP02_02 🔬 Subamostragem Espacial

Nesta atividade, você deve implementar a redução da resolução espacial de uma imagem através do processo de subamostragem.

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz original.
- Leia um valor inteiro f ($f \geq 1$), que representa o fator de amostragem.
- Leia os valores inteiros da matriz original.
- A nova imagem deve ser construída selecionando o pixel da posição $(f \cdot i, f \cdot j)$ da imagem original.
- Imprima a matriz resultante com as novas dimensões.
- Ver na Figure 2.12 uma simulação deste EP.

📌 **Importante**:

- **Dimensões Finais:** A imagem amostrada terá dimensões $\lceil L/f \rceil \times \lceil C/f \rceil$. No contexto de programação, isso equivale ao tamanho resultante de um fatiamento (slicing) com passo f.
- **Implementação:** Não utilize funções prontas de bibliotecas de processamento de imagens (como OpenCV ou PIL) para o redimensionamento. Implemente a lógica de seleção de pixels manualmente ou via fatiamento de matrizes.
- **Aliasing:** Note que este processo pode causar o efeito de *aliasing* (serrilhamento), onde detalhes finos são perdidos ou padrões indesejados aparecem.

2.12.2.1 🧠 Discretização do Espaço

A subamostragem reduz a resolução espacial de uma imagem, selecionando apenas um pixel a cada f pixels em cada direção. É o processo inverso da interpolação:

Parâmetro	Função	Efeito
Fator f	Salto de amostragem	Define o intervalo de seleção. Um fator 2 reduz a largura e altura pela metade.
Resolução	Densidade de pixels	Diminui a quantidade total de informação espacial da imagem.
Aliasing	Efeito colateral	Surgimento de padrões em escada ou blocos devido à perda de detalhes finos.

2.12.2.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

A terceira linha contém o fator `f`.

As linhas seguintes contêm os elementos da matriz $L \times C$.

Saída:

A matriz reduzida com as dimensões correspondentes ao fatiamento por `f`.

2.12.2.3 📌 Exemplos

Entrada	Saída	Observação
2 4 2 10 20 30 40 50 60 70 80	10 30	O fator 2 seleciona os pixels (0,0) e (0,2) da primeira linha. A segunda linha é ignorada.

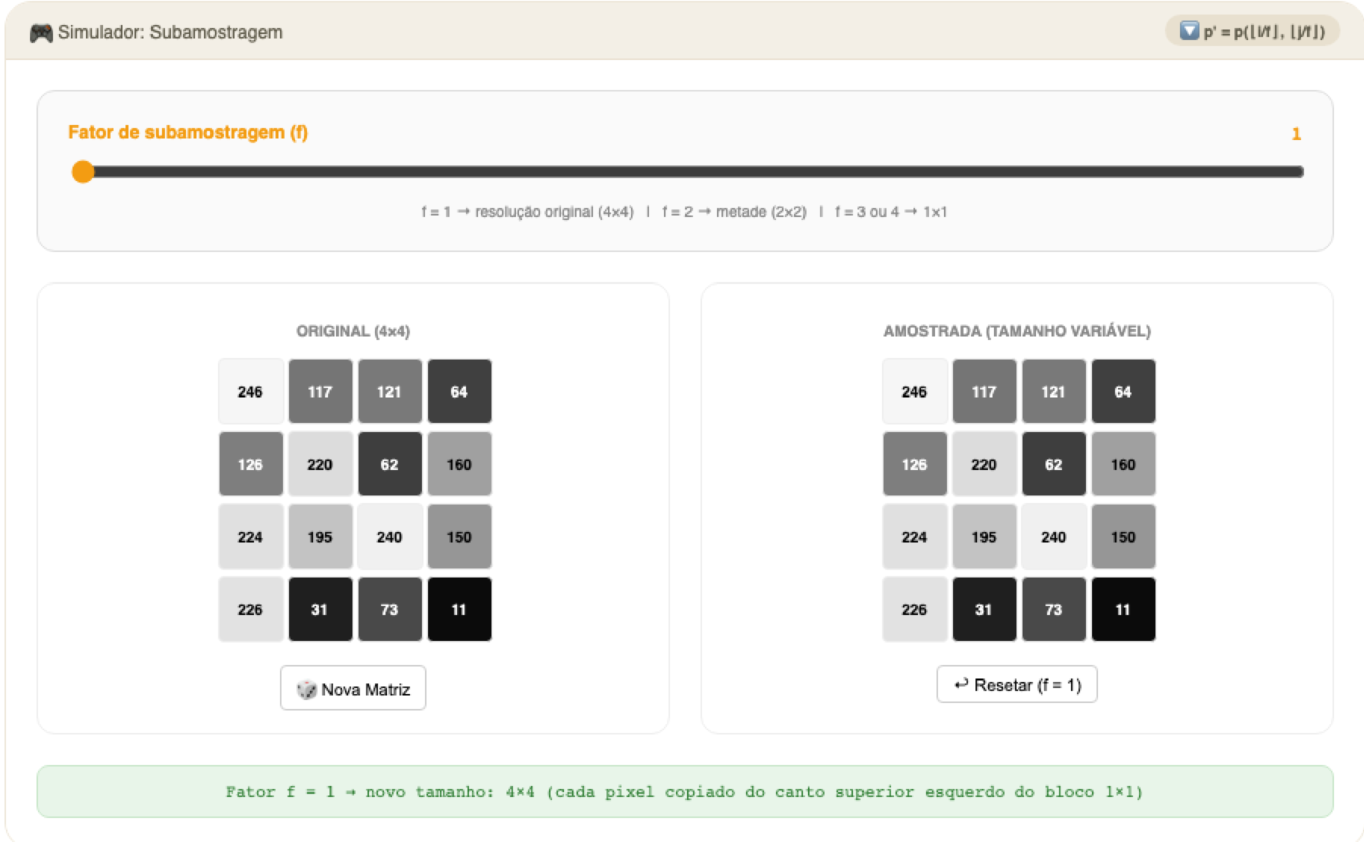


Figura 2.12: Simulador: Subamostragem

```
%%writefile EP02_02.py
# Código Python
```

```
Overwriting EP02_02.py
```

```
TestSuite("EP02_02.py").run()
```

2.12.3 EP02_03 🎨 Quantização de Níveis de Cinza

Nesta atividade, você deve implementar a quantização uniforme de uma imagem, reduzindo a quantidade de níveis de intensidade de cinza originais para uma nova escala baseada em um número menor de bits.

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz.
- Leia um inteiro k ($1 \leq k \leq 8$), representando o novo número de bits da imagem.
- Calcule o número de níveis ($N = 2^k$) e o tamanho do intervalo (passo).
- Para cada pixel p, calcule o novo valor p' mapeando-o para o nível discretizado correspondente.
- Imprima a matriz resultante com os mesmos valores de dimensões originais.
- Ver na Figure 2.13 uma simulação deste EP.

📌 **Importante**:

- **Posterização:** Ao reduzir drasticamente os níveis (ex: $k = 2$), você notará que degradês suaves se transformam em faixas abruptas de cor.
- **Cálculo do Passo:** O intervalo entre cada nível é definido por $passo = 256/2^k$.
- **Mapeamento:** Um método comum de quantização uniforme reconstrutiva é:

$$p' = \text{round}\left(\frac{p}{passo}\right) \times passo$$

- Certifique-se de que o valor final esteja clipado no intervalo $[0, 255]$.

2.12.3.1 🧠 Discretização da Amplitude

Enquanto a subamostragem lida com a resolução espacial, a quantização foca na precisão da cor (amplitude). Reduzir os bits significa simplificar a informação cromática:

Parâmetro	Função	Efeito
Bits (k)	Profundidade de cor	Define quantos tons diferentes a imagem pode ter (2^k).
Passo	Intervalo de tom	Espaçamento entre os níveis de cinza permitidos.
Posterização	Fenômeno visual	Transformação de variações contínuas em blocos de cor sólida.

2.12.3.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

A terceira linha contém o número de bits `k`.

As linhas seguintes contêm os elementos da matriz $L \times C$.

Saída:

A matriz transformada com os valores quantizados, mantendo o tamanho original $L \times C$.

2.12.3.3 📌 Exemplos

Entrada	Saída	Observação
1 4 2 0 80 170 255	0 1 2 3	Com $k = 2$, temos $2^2 = 4$ níveis: $0, 1, 2, 3$. O passo é $256/4 = 64$. Cada pixel é dividido pelo passo: $0 \to 0$, $80 \to 1$, $170 \to 2$, $255 \to 3$.
1 5 1 10 50 120 200 250	0 0 0 1 1	Com $k = 1$, temos $2^1 = 2$ níveis: 0 e 1. Passo = $256/2 = 128$. Pixels $< 128 \to 0$, pixels $\geq 128 \to 1$: $10 \to 0$, $50 \to 0$, $120 \to 0$, $200 \to 1$, $250 \to 1$.

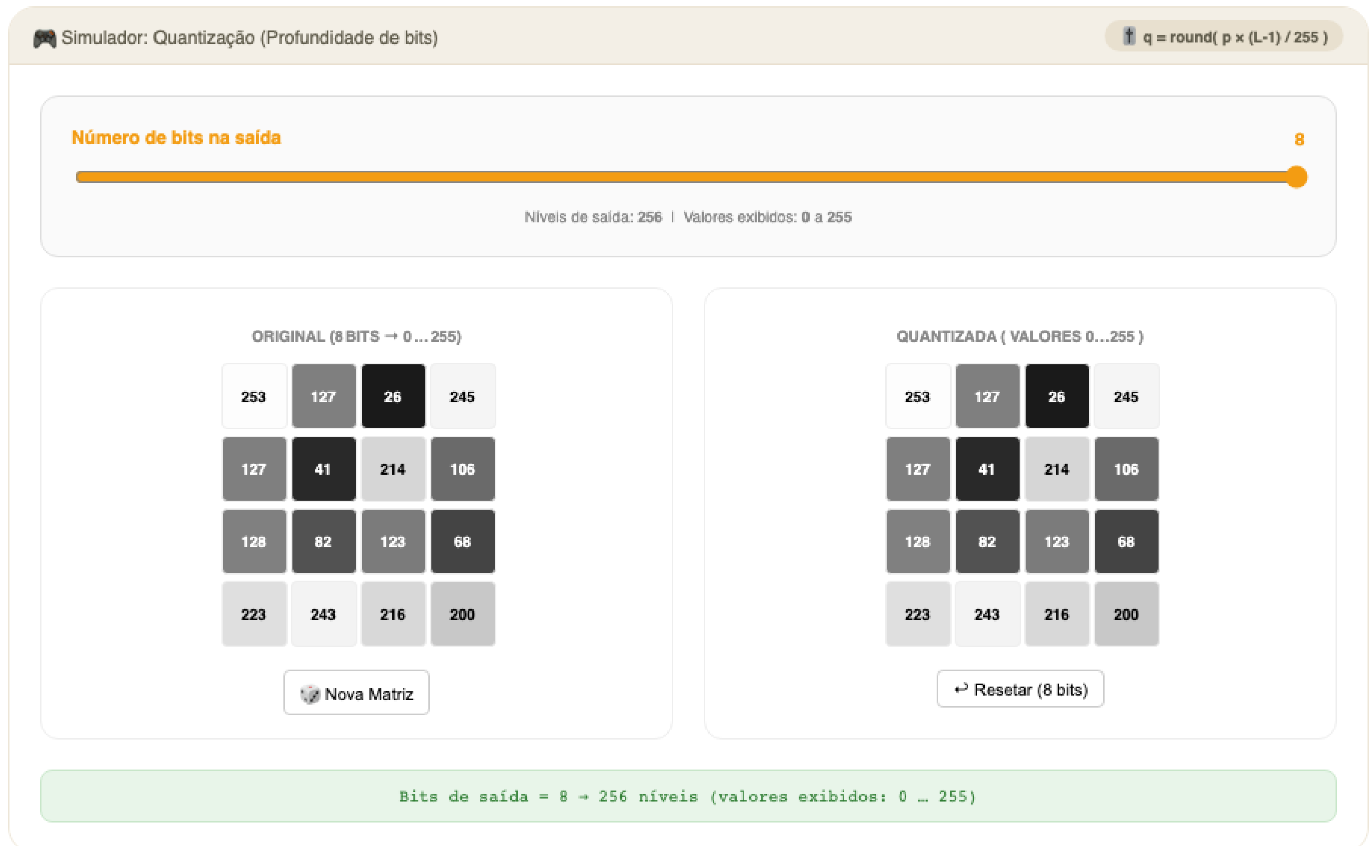


Figura 2.13: Simulador: Quantização

```
%%writefile EP02_03.py
# Código Python
```

```
Writing EP02_03.py
```

```
TestSuite("EP02_03.py").run()
```

2.12.4 EP02_04 📐 Transformada de Distância em Imagem Binária

Dada uma imagem binária onde pixels de valor **1** representam o *objeto* e pixels **0** representam o *fundo*, a distância de um pixel de fundo recebe a **menor distância ao pixel de objeto mais próximo**. Pixels de

objeto recebem distância **0**. Para simplificar, considere que a imagem tem **apenas um único objeto com um pixel de valor 1**.

Problema: Leia uma imagem binária $L \times C$ e uma métrica, e calcule essa distância simplificada aplicando uma das três fórmulas:

$$d_{\text{Euclidiana}} = \sqrt{(\Delta r)^2 + (\Delta c)^2}$$

$$d_{\text{City-block}} = |\Delta r| + |\Delta c|$$

$$d_{\text{Chessboard}} = \max(|\Delta r|,\ |\Delta c|)$$

onde Δr é a diferença de linhas e Δc a diferença de colunas entre dois pixels.

2.12.4.1 🖼️ Por que isso importa? - Aplicações da DT

A Transformada de Distância (DT) aparece em dezenas de pipelines de visão computacional:

Métrica	Complexidade	Aplicação típica
Euclidiana	🔴 $O(n^2)$ ingênuo	Esqueletização, matching de formas
City-block	🟡 $O(n)$ com 2 passes	Robótica, planejamento de caminho
Chessboard	🟢 $O(n)$ com 2 passes	Morfologia, dilatação/erosão

---

2.12.4.2 📌 Requisitos Técnicos

- **Entrada:** * Primeira linha: L e C (inteiros).
  - Segunda linha: nome da métrica (`euclidean`, `cityblock` ou `chessboard`).

  - Em seguida, a matriz binária $L \times C$ (valores 0 ou 1).
- **Pixels de objeto (1):** distância = 0 (ou 0.00 para euclidiana).

- **Pixels de fundo (0):** distância ao único pixel de objeto na imagem.

- **Arredondamento (euclidiana):** imprimir com **2 casas decimais** (formato `:.2f`). City-block e Chessboard produzem inteiros — imprimir sem decimais.

- **Saída:** valores separados por espaço, uma linha por linha da matriz.
- Ver na Figure 2.14 uma simulação deste EP.

2.12.4.3 📌 Exemplos

Entrada	Saída	Observação
4 4 chessboard 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0	2 2 2 2 2 1 1 1 2 1 0 1 2 1 1 1	A distância Chessboard é $\max(\|dx\|, \|dy\|)$. O único pixel objeto é $(2, 2) = 0$; os demais armazenam sua distância mínima até ele.

2.12.4.4 📌 Observações finais

- Como a imagem tem **apenas um objeto de um pixel**, a distância de cada pixel de fundo é simplesmente a distância desse pixel ao único ponto objeto.
- A implementação pode usar força bruta (percorrer todos os pixels da imagem e calcular a distância diretamente), pois L e C são pequenos nos casos de teste.
- Este problema é um aquecimento para a Transformada de Distância geral, que será trabalhada em capítulos posteriores com múltiplos objetos e algoritmos otimizados.

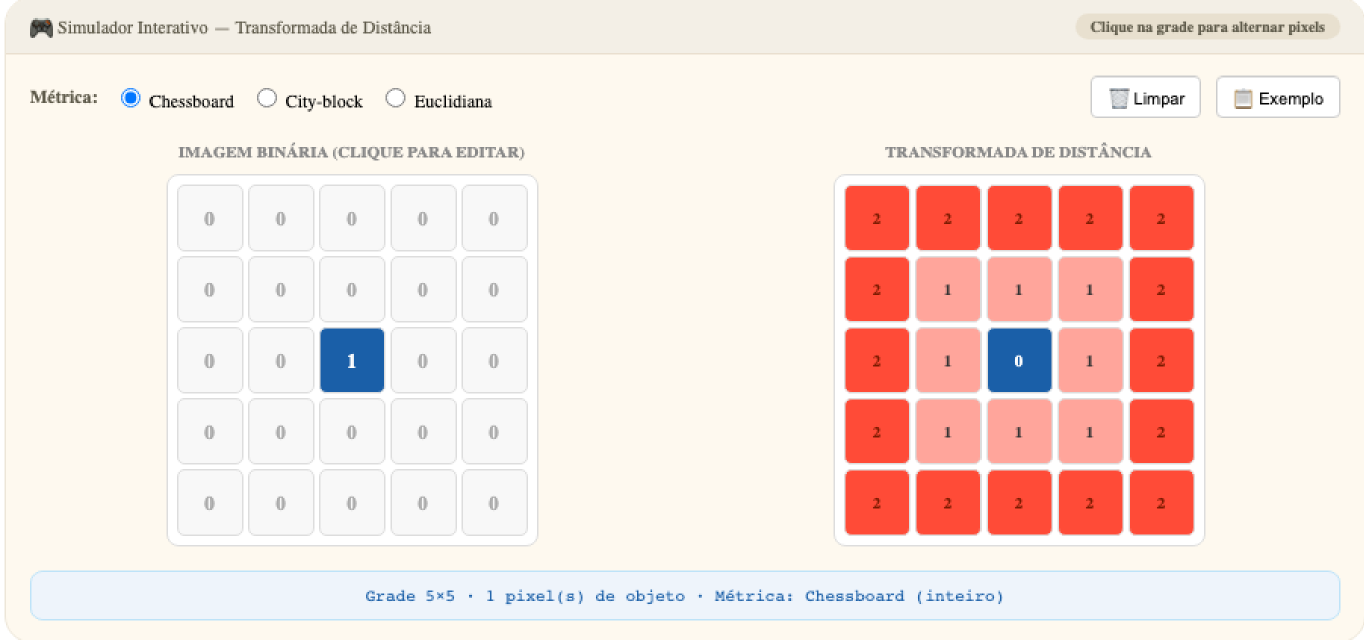


Figura 2.14: Simulador: Ajuste de Brilho e Contraste

```
%%writefile EP02_04.py
# Código Python
```

```
Writing EP02_04.py
```

```
TestSuite("EP02_04.py").run()
```

2.12.5 EP02_05 Translação de Imagem

Nesta atividade, você deve implementar o deslocamento espacial de uma imagem. A translação move cada pixel da imagem original para uma nova posição com base em um vetor de deslocamento.

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz.
- Leia dois inteiros t_x (deslocamento horizontal) e t_y (deslocamento vertical).
- Leia os valores inteiros da matriz original.
- Calcule a nova posição (x', y') para cada pixel (x, y) original.
- Imprima a matriz resultante com as mesmas dimensões da original.
- Ver na Figure 2.15 uma simulação deste EP.

📌 **Importante**:

- **Preenchimento:** Pixels que "entram" na imagem devido ao deslocamento e não possuem correspondente na original devem ser preenchidos com **0** (preto).
- **Descarte:** Pixels que, após a translação, ficarem fora dos limites da matriz ($0 \ldots L-1$ ou $0 \ldots C-1$) devem ser ignorados.
- **Coordenadas:** Considere x como o índice da linha e y como o índice da coluna.

2.12.5.1 🧠 Deslocamento Espacial

Transladar uma imagem significa mover todos os seus pontos por uma distância fixa em direções especificadas. Matematicamente, usando coordenadas homogêneas, a operação é descrita como:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Que resulta nas equações simples:

- $x' = x + t_x$
- $y' = y + t_y$

2.12.5.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

A terceira linha contém os inteiros `tx` e `ty`.

As linhas seguintes contêm os elementos da matriz $L \times C$.

Saída:

A matriz resultante com as mesmas dimensões $L \times C$ após o deslocamento.

2.12.5.3 📌 Exemplos

Entrada	Saída	Observação
2 2 1 1 10 20 30 40	0 0 0 10	Deslocamento $(t_x = 1, t_y = 1)$: cada pixel move uma posição para baixo e para a direita. Só o pixel $(0,0) = 10$ cabe no destino $(1,1)$; os demais saem da imagem. As posições vazias são preenchidas com 0.
3 3 -1 0 1 2 3 4 5 6 7 8 9	0 1 2 0 4 5 0 7 8	Deslocamento $(t_x = -1, t_y = 0)$: cada pixel move uma posição para a esquerda. A primeira coluna fica vazia (0) e a última coluna original é descartada.

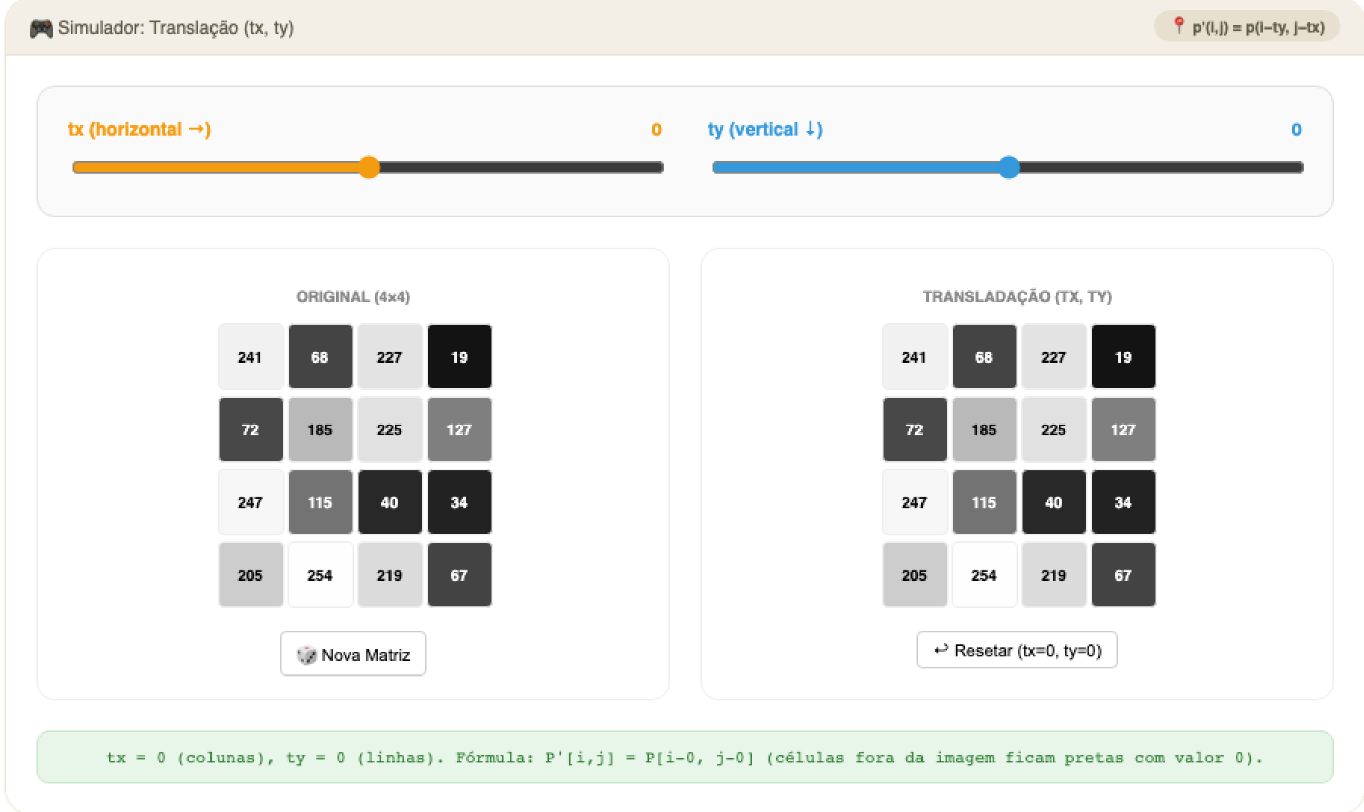


Figura 2.15: Simulador: Translação (tx, ty)

```
%%writefile EP02_05.py
# Código Python
```

```
Writing EP02_05.py
```

```
TestSuite("EP02_05.py").run()
```

2.12.6 EP02_06 🔄 Rotação de Imagem

Nesta atividade, você deve implementar a rotação de uma imagem em torno do seu centro geométrico. Esta operação requer o mapeamento de coordenadas e o uso de técnicas de interpolação para determinar os novos valores dos pixels.

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz.
- Leia um valor real θ (ângulo em graus) e uma string representando o **método** de interpolação (`nearest` ou `bilinear`).
- Leia os valores inteiros da matriz original.
- Realize a rotação em torno do centro da imagem $(L/2, C/2)$.
- Imprima a matriz resultante com as mesmas dimensões da original.
- Ver na Figure 2.16 uma simulação deste EP.

📌 **Importante**:

- **Mapeamento Inverso:** Para evitar "buracos" na imagem final, percorra cada pixel (x', y') da imagem de destino e calcule sua posição correspondente (x, y) na imagem original usando a matriz de rotação inversa.
- **Interpolação:**
- `nearest`: Atribui o valor do pixel mais próximo da coordenada calculada.
- `bilinear`: Calcula uma média ponderada baseada nos 4 vizinhos mais próximos.
- **Bordas:** Pixels cuja origem (x, y) caia fora dos limites da imagem original devem ser preenchidos com **0**.

2.12.6.1 🧠 Transformação por Ângulo

A rotação de um ponto (x, y) em relação à origem por um ângulo θ é dada pela matriz de transformação. Para rotacionar em torno de um centro (x_c, y_c), primeiro transladamos o centro para a origem, rotacionamos e transladamos de volta:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & x_c \\ \sin\theta & \cos\theta & y_c \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - x_c \\ y - y_c \\ 1 \end{bmatrix}$$

Dica: Use o mapeamento inverso para garantir que todos os pixels da imagem de saída sejam preenchidos corretamente.

2.12.6.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

A terceira linha contém o ângulo `theta` (em graus) e o método `interp` (`nearest` ou `bilinear`).

As linhas seguintes contêm os elementos da matriz $L \times C$.

Saída:

A matriz rotacionada com `L` linhas e `C` colunas.

2.12.6.3 📌 Exemplos

Entrada	Saída	Observação
2 2 90 nearest 1 2 3 4	3 1 4 2	Rotação de 90° horário: a coluna 0 vira a linha 0 (de baixo para cima). $(0,0) = 1 \to (1,0)$, $(1,0) = 3 \to (0,0)$, $(0,1) = 2 \to (1,1)$, $(1,1) = 4 \to (0,1)$.
3 3 45 bilinear 0 0 0 0 255 0 0 0 0	0 180 0 180 255 180 0 180 0	Rotação de 45°: o pixel central permanece 255; os vizinhos diretos recebem valor interpolado ≈ 180 por bilinear; os cantos permanecem 0.

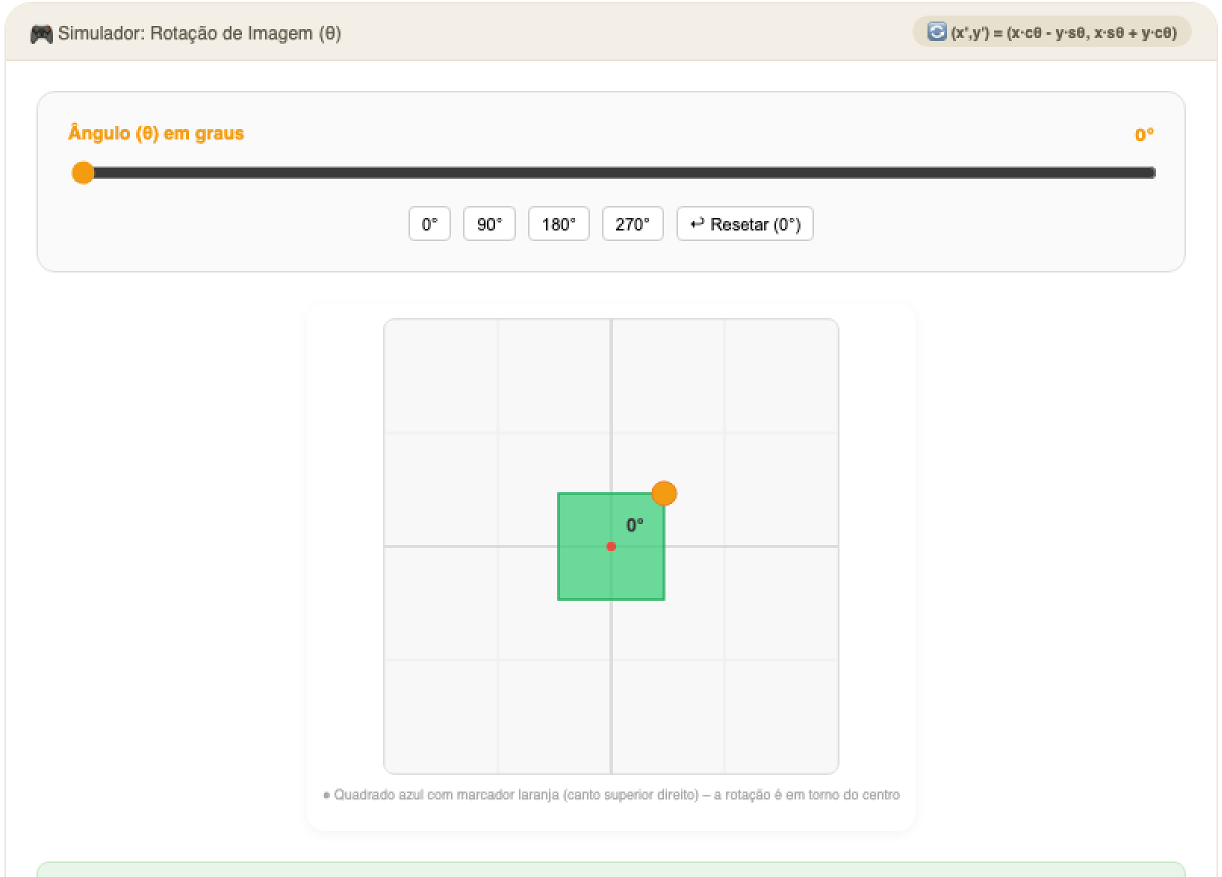


Figura 2.16: Simulador: Rotação (ângulo)

```
%%writefile EP02_06.py
# Código Python
```

```
Writing EP02_06.py
```

```
TestSuite("EP02_06.py").run()
```

2.12.7 EP02_07 🔍 Redimensionamento (Escala)

Nesta atividade, você deve implementar o redimensionamento de uma imagem utilizando fatores de escala. Diferente da subamostragem simples, aqui utilizaremos técnicas de interpolação para permitir tanto a ampliação quanto a redução da imagem.

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz original.
- Leia dois valores reais s_x (escala nas linhas) e s_y (escala nas colunas).
- Leia uma string representando o **método** de interpolação (`nearest` ou `bilinear`).
- Leia os valores inteiros da matriz original.
- Calcule as novas dimensões: $L' = \text{round}(L \times s_x)$ e $C' = \text{round}(C \times s_y)$.
- Imprima a matriz resultante com as novas dimensões.
- Ver na Figure 2.17 uma simulação deste EP.

📌 **Importante**:

- **Mapeamento Inverso:** Para cada pixel (x', y') da imagem de destino, encontre a posição correspondente na origem usando $(x, y) = (x'/s_x, y'/s_y)$.
- **Interpolação:**
- `nearest`: Seleciona o valor do pixel mais próximo (arredondamento das coordenadas).
- `bilinear`: Realiza uma interpolação linear dupla entre os quatro pixels vizinhos mais próximos na imagem original.
- **Bordas:** Certifique-se de que o mapeamento não tente acessar índices fora do intervalo $[0, L-1]$ e $[0, C-1]$.

2.12.7.1 🧠 Interpolação para Ampliação/Redução

Redimensionar uma imagem por fatores (s_x, s_y) exige o preenchimento de vazios (na ampliação) ou a fusão de informações (na redução). O método de interpolação define a qualidade visual do resultado:

Método	Funcionamento	Efeito Visual
Nearest	Pega o valor do vizinho mais próximo.	Rápido, mas gera efeito “pixelado” ou blocos.
Bilinear	Média ponderada dos 4 vizinhos (2×2).	Suaviza a imagem, reduzindo o serrilhamento.

2.12.7.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

A terceira linha contém os fatores `sx` e `sy`.

A quarta linha contém o método `interp` (`nearest` ou `bilinear`).

As linhas seguintes contêm os elementos da matriz $L \times C$.

Saída:

A matriz redimensionada com dimensões $L' \times C'$.

2.12.7.3 📌 Exemplos

Entrada	Saída	Observação
2 2 2.0 2.0 nearest 1 2 3 4	1 1 2 2 1 1 2 2 3 3 4 4 3 3 4 4	Ampliação 2×: cada pixel original é replicado em um bloco 2×2. A imagem 2×2 vira 4×4.
2 2 0.5 0.5 nearest 10 20 30 40	10	Redução 0.5×: a imagem 2×2 vira 1×1. Com nearest, o único pixel de saída amostra a posição $(0, 0) = 10$.

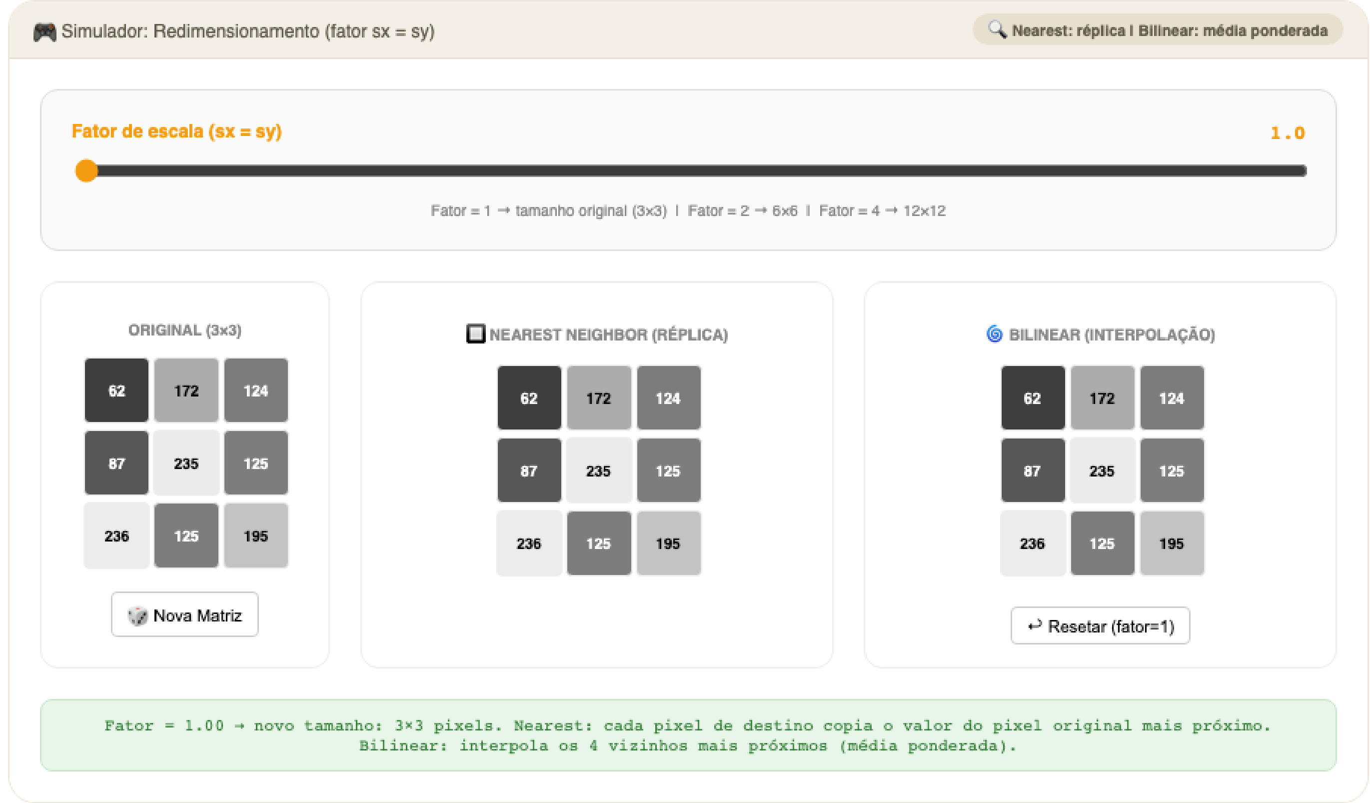


Figura 2.17: Simulador: Redimensionamento (Nearest vs Bilinear)

```
%%writefile EP02_07.py
# Código Python
```

```
Writing EP02_07.py
```

```
TestSuite("EP02_07.py").run()
```

2.12.8 EP02_08 🔀 Cisalhamento (Shear)

Nesta atividade, você deve implementar a transformação de cisalhamento em uma imagem. O cisalhamento é uma transformação afim que desloca cada ponto em uma direção fixada, por um valor proporcional à sua distância de uma reta paralela a essa direção, resultando em um efeito de inclinação.

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz.
- Leia dois valores reais sh_x (cisalhamento horizontal) e sh_y (cisalhamento vertical).
- Leia uma string representando o **método** de interpolação (`nearest` ou `bilinear`).
- Leia os valores inteiros da matriz original.
- Aplique a transformação mantendo o tamanho original da imagem (cortando o que ultrapassar os limites).
- Imprima a matriz resultante com as dimensões $L \times C$.
- Ver na Figure 2.18 uma simulação deste EP.

📌 **Importante**:

- **Mapeamento Inverso:** Para cada pixel (x', y') da imagem de destino, calcule a posição correspondente na origem (x, y) utilizando a matriz de cisalhamento inversa.
- **Preenchimento:** Coordenadas que resultarem em posições fora da matriz original devem ser preenchidas com **0**.
- **Coordenadas:** Para fins desta implementação, considere x como o índice da linha e y como o índice da coluna.

2.12.8.1 🧠 Distorção Afim

O cisalhamento altera a geometria da imagem inclinando seus eixos. A relação entre as coordenadas originais (x, y) e as transformadas (x', y') é dada por:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Isso resulta nas equações:

- $x' = x + sh_x \cdot y$
- $y' = y + sh_y \cdot x$

2.12.8.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

A terceira linha contém os fatores `shx` e `shy`.

A quarta linha contém o método `interp` (`nearest` ou `bilinear`).

As linhas seguintes contêm os elementos da matriz $L \times C$.

Saída:

A matriz transformada com as mesmas dimensões $L \times C$.

2.12.8.3 📌 Exemplos

Entrada	Saída	Observação
3 3 0.5 0.0 nearest 10 20 30 40 50 60 70 80 90	10 20 30 0 40 50 0 0 70	Cisalhamento horizontal: linha i desloca $\lfloor i \cdot 0.5 \rfloor$ pixels. Linha $0 \to 0$px, linha $1 \to 0$px, linha $2 \to 1$px. Os pixels deslocados para fora são descartados e as posições vazias preenchidas com 0.
2 2 0.0 1.0 nearest 10 20 30 40	10 0 30 20	Cisalhamento vertical: coluna j desloca $\lfloor j \cdot 1.0 \rfloor$ pixels para baixo. Coluna $0 \to 0$px (inalterada), coluna $1 \to 1$px: 20 desce para $(1, 1)$ e $(0, 1)$ fica 0.

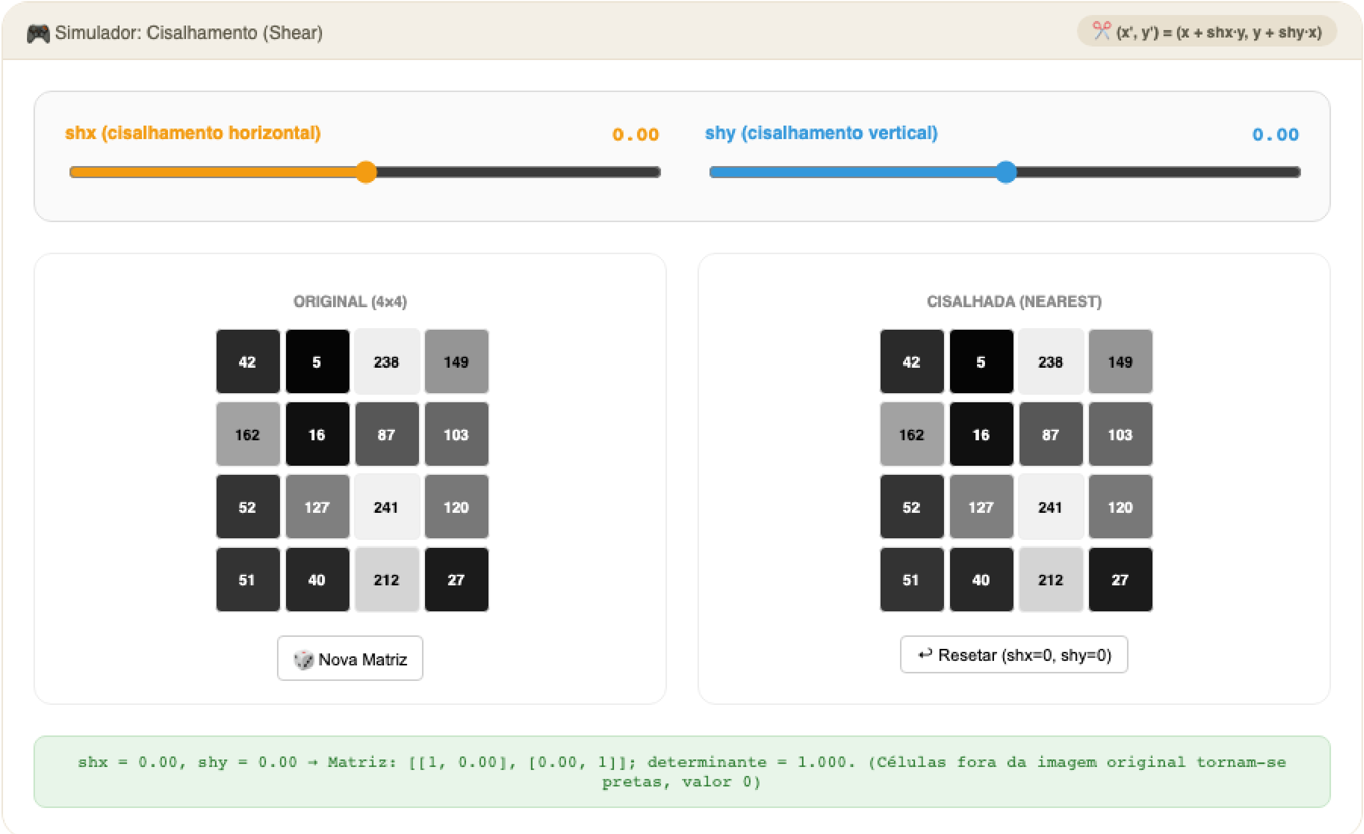


Figura 2.18: Simulador: Cisalhamento (shx, shy)

```
%%writefile EP02_08.py
# Código Python
```

```
Writing EP02_08.py
```

```
TestSuite("EP02_08.py").run()
```

2.12.9 EP02_09 🧩 Transformação Afim Genérica

Nesta atividade, você deve implementar uma transformação afim arbitrária em uma imagem. Esta operação é a generalização de todas as transformações lineares (escala, rotação, cisalhamento) combinadas com a translação, permitindo manipulações geométricas complexas através de uma única matriz.

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz.
- Leia **seis valores reais** (a, b, t_x, c, d, t_y) que compõem a matriz de transformação afim 2×3.
- Leia uma string representando o **método** de interpolação (`nearest` ou `bilinear`).
- Leia os valores inteiros da matriz original.
- Aplique a transformação mantendo o tamanho original $L \times C$.
- Imprima a matriz resultante.
- Ver na Figure 2.19 uma simulação deste EP.

📌 **Importante**:

- **Mapeamento Inverso:** Para calcular o valor de cada pixel na imagem de destino, você deve utilizar a **inversa** da matriz de transformação afim fornecida para encontrar a coordenada correspondente na imagem original.
- **Preenchimento:** Coordenadas calculadas que caiam fora dos limites $[0, L-1]$ e $[0, C-1]$ da imagem original devem resultar em um pixel de valor **0**.
- **Flexibilidade:** Esta implementação deve ser capaz de realizar qualquer uma das tarefas anteriores (translação, rotação, etc.) bastando alterar os parâmetros da matriz.

2.12.9.1 🧠 Combinação de Operações

A transformação afim preserva pontos, retas e planos. No processamento de imagens, ela mapeia a posição (x, y) para (x', y') seguindo o sistema:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

Ou, de forma compacta em coordenadas homogêneas:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2.12.9.2 📋 Tarefa (especificação para VPL)

Entrada:

A primeira linha contém `L`.

A segunda linha contém `C`.

A terceira linha contém seis floats: `a b tx c d ty`.

A quarta linha contém o método `interp` (`nearest` ou `bilinear`).

As linhas seguintes contêm os elementos da matriz $L \times C$.

Saída:

A matriz transformada com as dimensões originais $L \times C$.

2.12.9.3 📌 Exemplos

Entrada	Saída	Observação
2 2 1.0 0.0 0.5 0.0 1.0 0.5 bilinear 10 20 30 40	15 20 25 30	Translação fracionária ($t_x = 0.5, t_y = 0.5$): cada pixel de saída (i, j) amostra a posição $(i + 0.5,\ j + 0.5)$ da entrada via bilinear. Ex: $(0, 0)$ interpola os quatro vizinhos $\rightarrow$ 15.
3 3 2.0 0.0 0.0 0.0 2.0 0.0 nearest 1 2 3 4 5 6 7 8 9	1 1 2 1 1 2 4 4 5	Escala 2× via matriz afim ($a = 2, d = 2$): cada pixel de saída (i, j) amostra a posição $(2i, 2j)$ da entrada com nearest. Ex: $(0, 2) \rightarrow (0, 4)$ fora da imagem $\rightarrow$ nearest clipa para $(0, 2) = 3$... aguarda confirmação da lógica de borda.

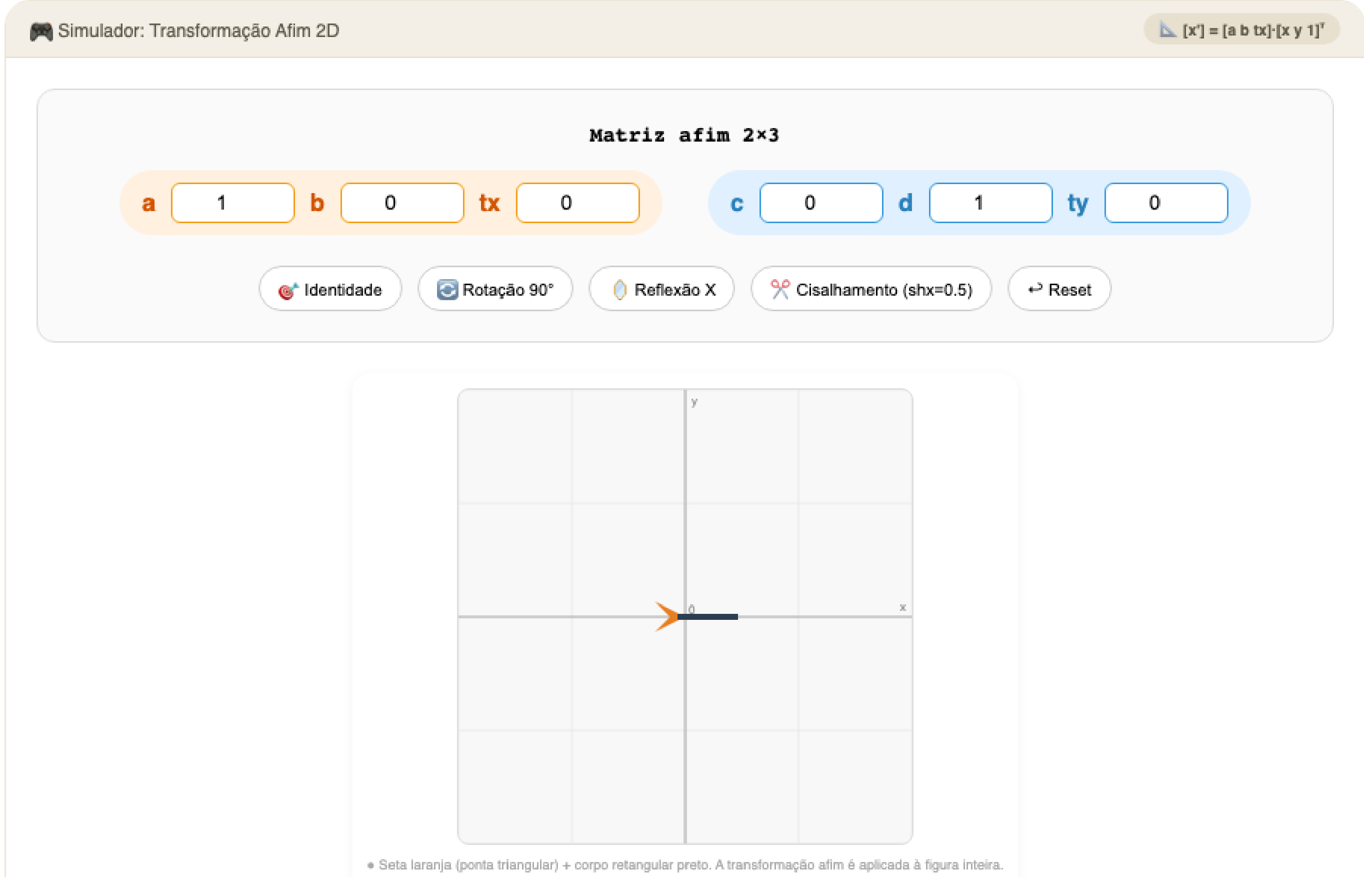


Figura 2.19: Simulador: Transformação Afim (matriz 2×3)

```
%%writefile EP02_09.py
# Código Python
```

```
Writing EP02_09.py
```

```
TestSuite("EP02_09.py").run()
```

2.12.10 EP02_10 🎯 Correção de Perspectiva (Homografia)

Nesta atividade, você deve implementar a transformação de perspectiva, também conhecida como homografia. Diferente das transformações afins, a perspectiva não preserva o paralelismo, permitindo "retificar" objetos inclinados, como documentos ou placas capturados em ângulos oblíquos.

- Leia dois inteiros **L** e **C**, representando as dimensões da matriz original.
- Leia **quatro pares de coordenadas** (x, y) representando os cantos do quadrilátero de origem (objeto distorcido).
- Leia **quatro pares de coordenadas** (x, y) representando os cantos do quadrilátero de destino (onde o objeto deve ser mapeado).
- Leia os valores da matriz original.
- Calcule a matriz de homografia 3×3 e aplique a transformação.
- Imprima a matriz resultante com as dimensões de saída especificadas.
- Ver na Figure 2.20 uma simulação deste EP.

📌 **Importante**:

- **Graus de Liberdade:** A homografia possui 8 graus de liberdade (o nono elemento da matriz 3×3 é uma constante de normalização, geralmente 1), exigindo no mínimo 4 pontos correspondentes para ser calculada.
- **Projeção:** Após multiplicar as coordenadas pela matriz, é necessário dividir os resultados x' e y' pela componente homogênea w para retornar ao plano 2D.
- **Uso de Bibliotecas:** Para esta tarefa, você pode utilizar as funções `cv2.getPerspectiveTransform` para obter a matriz e `cv2.warpPerspective` para aplicar a transformação, ou implementar o sistema linear e o mapeamento inverso manualmente para um desafio extra.

2.12.10.1 🧠 Deformação não-afim

Enquanto transformações afins mapeiam paralelogramos em paralelogramos, a homografia mapeia qualquer quadrilátero em outro quadrilátero. Isso é essencial para visão computacional:

Operação	Característica	Aplicação Típica
Homografia	Projeção em plano	Correção de documentos, escaneamento de placas.
Ponto de Fuga	Convergência de linhas	Reconstrução 3D a partir de imagens 2D.
Warping	Deformação de malha	Estabilização de vídeo e panoramas (stitching).

2.12.10.2 📌 Exemplos

Entrada	Saída	Observação
60 65 390 52 28 408 415 413 0 0 300 0 0 300 300 300	Matriz 300 × 300	As 4 primeiras linhas são os pontos de origem; as 4 seguintes são os destinos. Transforma o trapézio irregular em um quadrado perfeito, corrigindo a perspectiva da câmera.

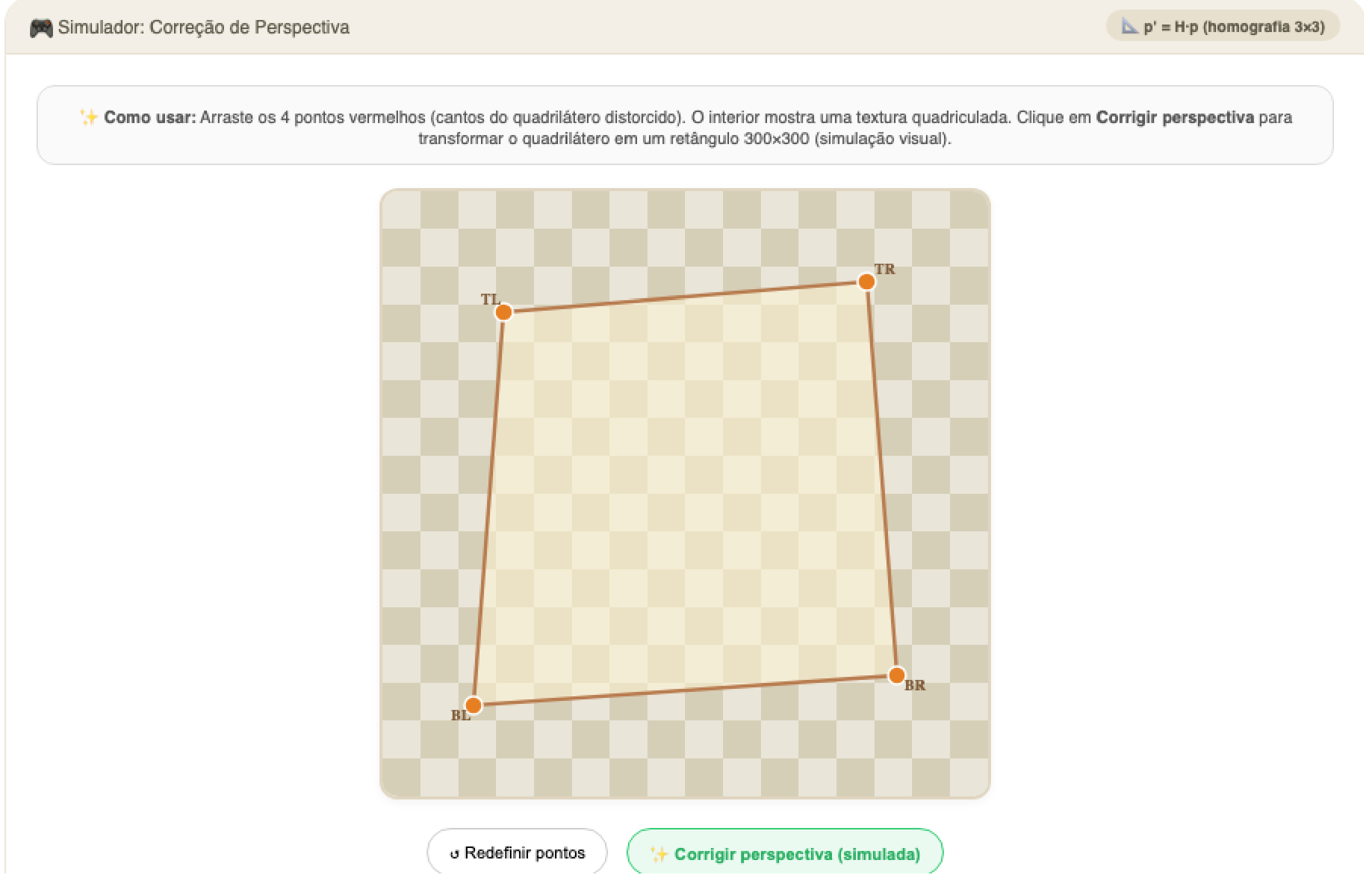


Figura 2.20: Simulador: Correção de Perspectiva (Homografia)

```
%%writefile EP02_10.py
# Código Python
```

```
Writing EP02_10.py
```

TestSuite("EP02_10.py").run()

2.12.11 EP02_11 🎯 Correção de Perspectiva em Imagem Real (Sudoku)

Nesta atividade, o objetivo é aplicar a transformação de perspectiva (homografia) em uma fotografia real. Diferente de exemplos sintéticos, você lidará com uma imagem de jornal inclinada, onde a grade do Sudoku não está perfeitamente retangular devido ao ângulo da captura.

- Baixe a imagem através da URL fornecida (utilizando o cabeçalho adequado para evitar erros de acesso).
- Aplique uma expansão (**padding**) na imagem original com `cv2.copyMakeBorder` para garantir que os vértices do objeto não sejam cortados durante o processamento.
- Identifique os **quatro pontos de canto** (x, y) da grade do Sudoku na imagem expandida (TL, TR, BL, BR).
- Defina os **quatro pontos de destino** correspondentes para mapear o tabuleiro em uma vista frontal quadrada (ex: 500×500).
- Utilize `cv2.getPerspectiveTransform` para calcular a matriz de homografia e `cv2.warpPerspective` para retificar a imagem.
- Exiba o resultado da imagem corrigida lado a lado com a original.
- Ver na Figure 2.22 uma simulação deste EP.

📌 **Importante**:

- **Ponto de Transdução:** Lembre-se que a formação desta imagem começou com a transdução da luz refletida pelo papel em sinais eletrônicos no sensor da câmera.
- **Mapeamento Inverso:** Funções como `warpPerspective` utilizam o mapeamento inverso com interpolação (geralmente bilinear) para evitar buracos na imagem de saída, calculando qual pixel da origem corresponde a cada posição do destino.
- **Créditos:** A imagem utilizada é **"Sudoku en periódico"** de Héctor Rodríguez, sob licença **CC BY 2.0**. A fonte original e dados de GPS estão disponíveis no material de apoio.

2.12.11.1 🧠 Contexto do Problema

A homografia é uma ferramenta poderosa na visão computacional para remover distorções de perspectiva. Ao contrário de transformações afins, ela permite que linhas que convergem para pontos de fuga sejam tornadas paralelas novamente:

Operação	Característica	Aplicação Típica
Homografia	Projeção entre planos	Retificação de documentos, leitura de QR Codes.
Padding	Expansão de bordas	Proteção de gradientes em vértices periféricos.
Warping	Reamostragem espacial	Correção de lentes e montagem de panoramas.

2.12.11.2 📌 Exemplos

Entrada (pts1 — origem)	Saída	Observação
120 95 480 75 100 510 500 495	Imagem 500×500 px	Os pontos de origem marcam os cantos externos da grade na imagem expandida. A homografia produz uma visão frontal e sem distorção do tabuleiro, facilitando a extração de dígitos por algoritmos de OCR.

2.12.11.3 Aquisição da imagem do sudoku e conversão para níveis de cinza

O processamento inicia-se com a leitura da imagem original e a extração de seus metadados EXIF, que revelam a geolocalização precisa da captura, conforme detalhado na Figure 2.21. Após a conversão para tons de cinza e o redimensionamento para uma matriz de 500×500 pixels, os dados são preparados para a retificação geométrica. Este processo de correção de perspectiva, essencial para eliminar deformações causadas pelo ângulo da câmera, pode ser explorado de forma interativa através da Figure 2.22, onde a aplicação da matriz de homografia permite obter uma visão frontal e regular da grade do Sudoku.

```
Dados de Aquisição:
{'Make': 'Panasonic', 'Model': 'DMC-FX12', 'DateTime': '2009:04:14 23:10:05'}

  GPS não veio no EXIF local.
  Consultando Wikimedia Commons API...
  GPS recuperado via API (extmetadata).

GPS Decimal : 38.090876, -0.654695
Google Maps : https://www.google.com/maps?q=38.090876,-0.654695

Tipo PIL: <class 'PIL.JpegImagePlugin.JpegImageFile'>
Dimensões PIL (x,y): (3072, 2304)

Tipo NumPy: <class 'numpy.ndarray'>
Dimensões NumPy [y,x,c]: (2304, 3072, 3)

Pixel Pillow (0,0): (28, 25, 8)
Pixel NumPy  [0,0]: [28 25  8]

  Arquivo salvo: sudoku.png
```

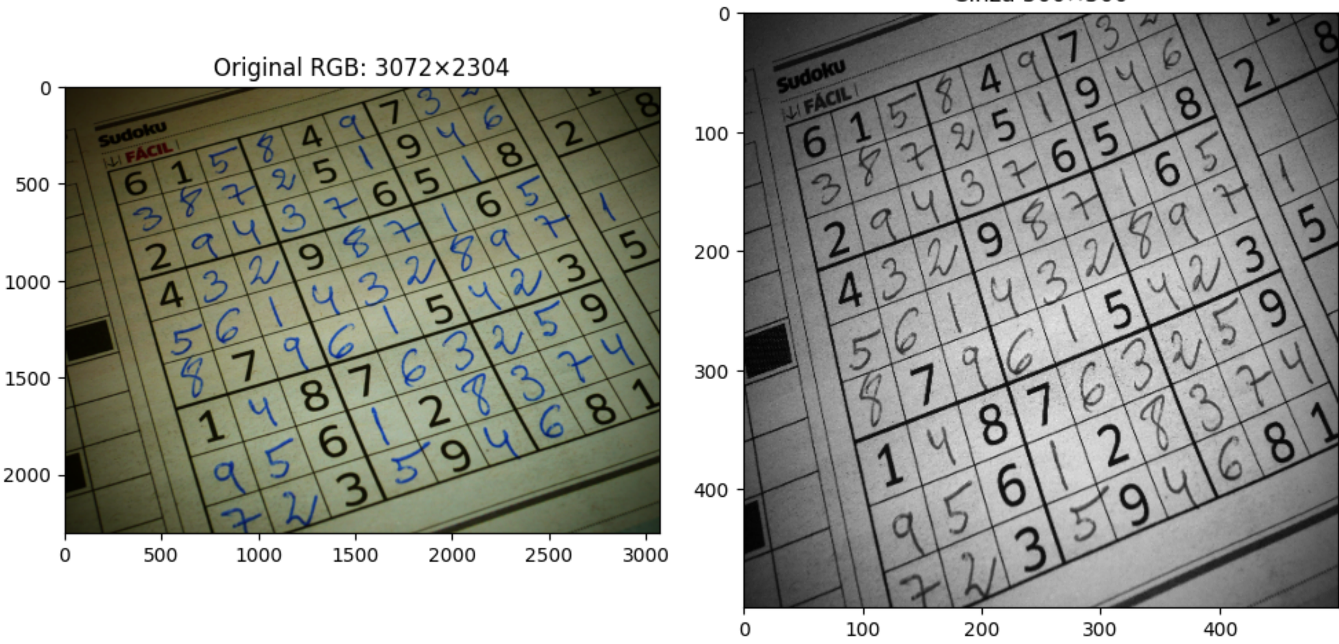


Figura 2.21: Aquisição da imagem de um Sudoku à esquerda. À direita, conversão para tons de cinza e redimensionamento. Crédito: Héctor Rodríguez de Guardamar, Espanha (CC BY 2.0).

```
%%writefile EP02_11.py
# Código Python
```

```
Writing EP02_11.py
```

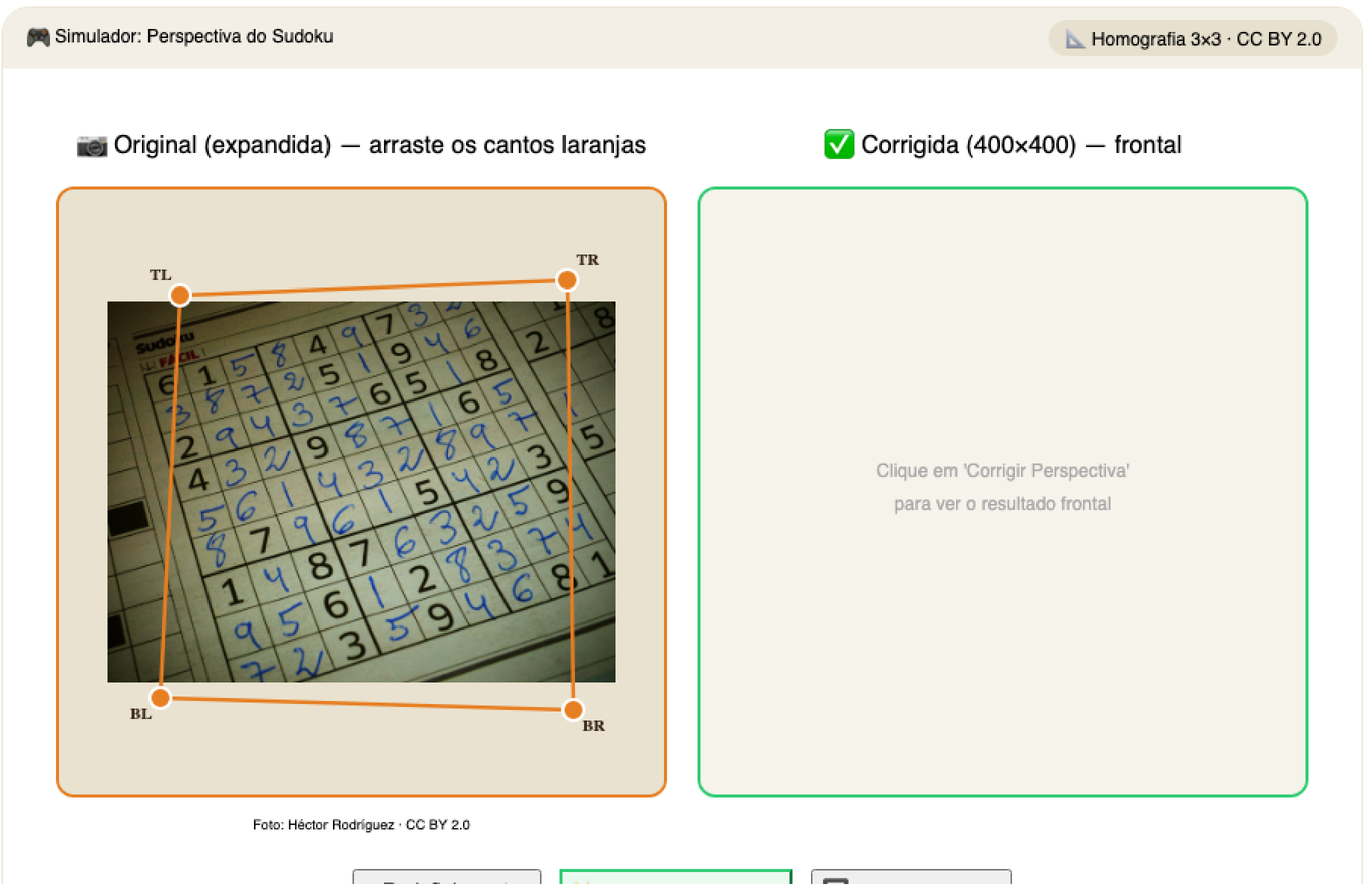


Figura 2.22: Simulador: correção de perspectiva do Sudoku.

```
TestSuite("EP02_11.py").run()
```

Capítulo 3

Operações Espaciais: Intensidade, Histograma e Filtragem

Este capítulo aprofunda o processamento de imagens no domínio espacial, partindo da manipulação direta de pixels e histogramas para o realce de contraste, até a aplicação de filtros locais por convolução para suavização, redução de ruído e detecção de bordas. O objetivo é desenvolver a intuição matemática e computacional que sustenta grande parte dos algoritmos modernos de Visão Computacional.

3.1 Objetivos

Ao final deste capítulo, você será capaz de:

- **Manipular intensidade e pixels:** Executar operações aritméticas saturadas (`mm.addm`, `mm.subm`) e lógicas bit a bit (`mm.band`, `mm.bor`, `mm.bnot`) para combinação e seleção de regiões de interesse (ROI), e aplicar *alpha blending* (`mm.blend`) para fusão ponderada de imagens;
- **Processar histogramas:** Interpretar o histograma como diagnóstico tonal, aplicar equalização global (`mm.equalize`) e adaptativa (CLAHE), e realizar especificação de histograma para transferência de perfil tonal entre imagens;
- **Compreender fundamentos espaciais:** Entender vizinhança, *padding* de borda, e a diferença entre correlação cruzada (`mm.conv`) e convolução — incluindo por que kernels assimétricos como Sobel produzem resultados distintos nas duas operações;
- **Aplicar filtragem de suavização:** Utilizar o filtro de média (`mm.conv` com kernel uniforme) e o filtro Gaussiano (`cv2.GaussianBlur`) para redução de ruído, compreendendo a vantagem da ponderação radial e da separabilidade Gaussiana;
- **Aplicar filtragem de realce:** Usar o Laplaciano (w_4 e w_8) para realce isotrópico de bordas, o operador de Sobel para gradiente direcional (G_x, G_y, magnitude e direção), e o *Unsharp Masking* para amplificação controlada de alta frequência pelo parâmetro k;
- **Utilizar filtros de ordem:** Aplicar o filtro da mediana (`cv2.medianBlur`) para remoção eficaz de ruído sal e pimenta, compreendendo por que sua natureza não linear e robustez a outliers o tornam superior aos filtros lineares nesse cenário;
- **Resolver problemas práticos:** Encadear técnicas em pipelines de pré-processamento (ex.: CLAHE → Gaussiano → Canny) e utilizar as funções da biblioteca `morph.py` (`mm.conv`, `mm.histImg`, `mm.equalize`, `mm.drawImageKernel`) para análise e visualização didática de cada etapa.

3.2 Operações em Nível de Intensidade

O nível mais elementar de processamento de imagens atua diretamente sobre os valores dos pixels, sem considerar vizinhança. Essas operações — chamadas de **transformações de ponto** (*point operations*) — são as mais rápidas computacionalmente e formam a base para técnicas mais complexas.

Formalmente, uma transformação de ponto pode ser descrita como:

$$g(x, y) = T[f(x, y)] \tag{3.1}$$

onde $f(x, y)$ é a imagem de entrada, $g(x, y)$ é a saída e T é uma função aplicada a cada pixel individualmente.

```
import os, importlib, urllib.request
import numpy as np
import matplotlib.pyplot as plt
import cv2

# Baixar morph.py se necessário
BASE_URL = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph"
for f in ["morph.py"]:
    if not os.path.exists(f):
        urllib.request.urlretrieve(f"{BASE_URL}/{f}", f)

import morph
importlib.reload(morph)
from morph import mm

print("  Ambiente pronto")
```

```
  Ambiente pronto
```

Como objeto de estudo ao longo deste capítulo, utilizaremos as imagens de vida selvagem apresentadas nas Figure 3.1 e Figure 3.3. A partir delas, analisaremos a estrutura de matrizes multidimensionais, a conversão para tons de cinza e a extração de metadados espaciais obtidos diretamente do cabeçalho EXIF dos arquivos originais.

```
import os
from PIL.ExifTags import TAGS

# Fonte : https://commons.wikimedia.org/wiki/File:Carlos_Guilherme_Rodrigues_(76515283).jpeg
# Mapa  : https://www.google.com/maps?q=-26.204734,28.226624

base    = "https://upload.wikimedia.org/wikipedia/commons"
arquivo = "Carlos_Guilherme_Rodrigues_%2876515283%29.jpeg"
url     = f"{base}/9/9b/{arquivo}"
caminho = "imagens/mandrill-exif.jpg"

# 1. Leitura com cache local
if not os.path.exists(caminho):
    os.makedirs("imagens", exist_ok=True)
    img_obj = mm.read(url, pil=True)
    mm.write(img_obj, caminho)
else:
    img_obj = mm.read(caminho, pil=True)

img_color = np.array(img_obj)
img_gray  = mm.gray(img_color)

# 2. Extração e conversão de GPS (Tag 34853)
exif = img_obj._getexif()
if exif and (gps := exif.get(34853)):
    to_dec = lambda dms, ref: float(-(dms[0]+dms[1]/60+dms[2]/3600) if ref in 'SW'
                                    else (dms[0]+dms[1]/60+dms[2]/3600))
    lat, lon = to_dec(gps[2], gps[1]), to_dec(gps[4], gps[3])
    print(f"GPS Decimal: {lat:.6f}, {lon:.6f}")
    print(f"Maps: https://www.google.com/maps/search/?api=1&query={lat},{lon}")
```

```
# 3. Diagnóstico de tipos, dimensões e acesso a pixels
print(f"\nTipo PIL  : {type(img_obj)}   | Dimensões (x,y): {img_obj.size}")
print(f"Tipo NumPy: {type(img_color)} | Dimensões [y,x,c]: {img_color.shape}")

# 4. Exibição
mm.show(img_color, scale=90)
```

```
Tipo PIL  : <class 'PIL.JpegImagePlugin.JpegImageFile'>   | Dimensões (x,y): (960, 540)
Tipo NumPy: <class 'numpy.ndarray'> | Dimensões [y,x,c]: (540, 960, 3)
```

Figura 3.1: *Mandrill* (*Mandrillus sphinx*) fotografado em ambiente natural na África do Sul. A imagem possui resolução de 500 px e contém metadados EXIF com coordenadas GPS aproximadas (-26.20473, 28.22662). Crédito: Carlos Guilherme Rodrigues (CC BY-SA 3.0).

3.2.1 Operações Aritméticas

Operações aritméticas entre imagens são amplamente usadas em PDI para combinar, comparar ou realçar informações. A **subtração de imagens** é especialmente poderosa para detectar diferenças entre dois quadros — por exemplo, na remoção de fundo estático em câmeras de vigilância:

$$g(x, y) = f_1(x, y) - f_2(x, y) \tag{3.2}$$

A **adição saturada** limita o resultado ao intervalo $[0, 255]$: valores acima de 255 são fixados em 255, evitando o *overflow* silencioso do tipo `uint8` (ex.: $200 + 100 = 44$ em vez de 300). A **subtração saturada** aplica o mesmo princípio pelo lado inferior: valores negativos são fixados em 0.

> ⚠ Saturação e *overflow*
>
> Operações aritméticas em `uint8` sofrem *overflow* silencioso: $200 + 100 = 44$ (não 300). As funções `mm.addm` e `mm.subm` usam `cv2.add` e `cv2.subtract`, que realizam saturação automática. Para o *blending*, converta para `float32` antes de operar e aplique `np.clip(..., 0, 255).astype(np.uint8)` ao final, pois a operação envolve pesos fracionários.

A Figure 3.2 demonstra adição de uma constante (clareamento) e subtração de uma constante (escurecimento com saturação em 0).

```
fundo = 60

img_add = mm.addm(img_gray, fundo)
```

```
img_sub = mm.subm(img_gray, fundo)

mm.show(
    [img_gray, img_add, img_sub],
    titles=["Original", "addm (+60)", "subm (-60)"],
    cols=3
)
```

Figura 3.2: Operações aritméticas saturadas: adição de constante (clareamento) e subtração de constante (escurecimento com saturação em 0).

3.2.2 Mistura Ponderada (*Alpha Blending*)

A **mistura ponderada** (*alpha blending*) combina duas imagens utilizando pesos complementares α e $(1-\alpha)$:

$$g(x, y) = \alpha\, f_1(x, y) + (1-\alpha)\, f_2(x, y), \quad \alpha \in [0, 1] \tag{3.3}$$

Quando $\alpha = 1$, obtém-se apenas a imagem f_1; quando $\alpha = 0$, apenas f_2. Valores intermediários produzem uma transição suave entre ambas, sendo amplamente utilizados em composição de imagens, sobreposição de camadas, marcas d'água e efeitos de fusão visual.

Para que a combinação produza um resultado coerente, é necessário alinhar previamente as regiões de interesse das imagens. Na Figure 3.4, utiliza-se um recorte do rosto do leopardo:

```
leo = img_gray_leo[250:-300, 100:-200]
```

e um recorte central da região facial do mandril:

```
mandrill = img_gray[100:400, 380:530]
```

As duas regiões são escolhidas de forma que olhos e estrutura facial permaneçam aproximadamente alinhados. Em seguida, o recorte do leopardo é redimensionado para coincidir com as dimensões do mandril antes da aplicação da mistura ponderada.

A operação é realizada em `float32` para evitar problemas de *overflow* durante as operações aritméticas, seguida de *clipping* e conversão final para `uint8`.

```
import os
from PIL.ExifTags import TAGS

base    = "https://upload.wikimedia.org/wikipedia/commons"
arquivo = "Leopard_%28Panthera_pardus%29_portrait.jpg"
url     = f"{base}/9/92/{arquivo}"
caminho = "imagens/leopardo.jpg"

# 1. Leitura com cache local
if not os.path.exists(caminho):
    os.makedirs("imagens", exist_ok=True)
    img_obj = mm.read(url, pil=True)
```

```
    mm.write(img_obj, caminho)
else:
    img_obj = mm.read(caminho, pil=True)

img_numpy    = np.array(img_obj)
img_gray_leo = mm.gray(img_numpy)

# 2. Extração e conversão de GPS (Tag 34853)
exif = img_obj._getexif()
if exif and (gps := exif.get(34853)):
    to_dec = lambda dms, ref: float(
        -(dms[0]+dms[1]/60+dms[2]/3600) if ref in 'SW'
        else (dms[0]+dms[1]/60+dms[2]/3600)
    )
    lat, lon = to_dec(gps[2], gps[1]), to_dec(gps[4], gps[3])
    print(f"GPS Decimal: {lat:.6f}, {lon:.6f}")
    print(f"Maps: https://www.google.com/maps/search/?api=1&query={lat},{lon}")

# 3. Diagnóstico de tipos, dimensões e acesso a pixels
print(f"\nTipo PIL   : {type(img_obj)}   | Dimensões (x,y): {img_obj.size}")
print(f"Pillow (0,0): {img_obj.getpixel((0, 0))}")
print(f"Tipo NumPy: {type(img_numpy)} | Dimensões [y,x,c]: {img_numpy.shape}")
print(f"NumPy [0,0]: {img_numpy[0, 0]}")

# 4. Exibição
mm.show(img_numpy)
```

```
GPS Decimal: -19.140200, 23.814872
Maps: https://www.google.com/maps/search/?api=1&query=-19.1402,23.814872222222224

Tipo PIL  : <class 'PIL.JpegImagePlugin.JpegImageFile'>   | Dimensões (x,y): (1242, 1324)
Pillow (0,0): (191, 135, 88)
Tipo NumPy: <class 'numpy.ndarray'> | Dimensões [y,x,c]: (1324, 1242, 3)
NumPy [0,0]: [191 135  88]
```

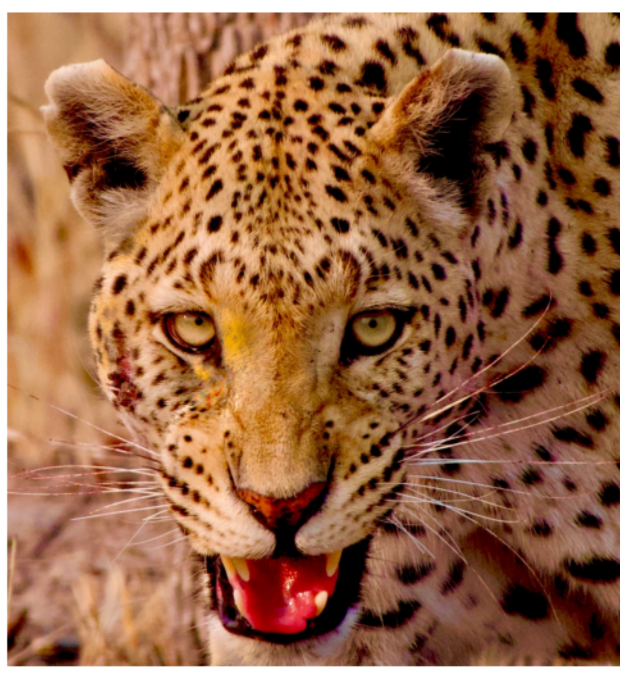


Figura 3.3: Retrato de um leopardo (*Panthera pardus*) em ambiente natural. Crédito: C. Brück (CC BY-SA 4.0).

```
leo = img_gray_leo[250:-300, 100:-200]
mandrill = img_gray[100:400, 380:530]
img_leopardo = mm.resize(leo, (mandrill.shape[1], mandrill.shape[0]), method='bilinear')

alphas = [1.0, 0.8, 0.6, 0.4, 0.2, 0.0]
imgs_blend   = []
titles_blend = []

for a in alphas:
    imgs_blend.append(mm.blend(mandrill, img_leopardo, alpha=a))
    titles_blend.append(f" ={a:.1f}")

mm.show(imgs_blend, titles=titles_blend, cols=6, figsize=(18, 16))
```

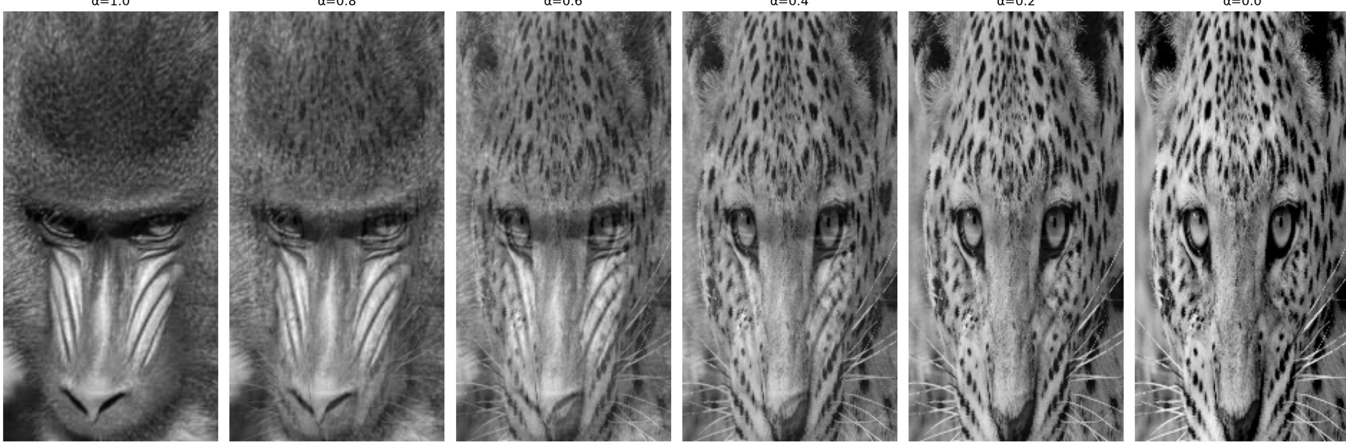


Figura 3.4: *Alpha blending* entre recortes alinhados de mandrill e do leopardo (Figure 3.3) para diferentes valores de . Em =1 vê-se apenas mandrill; em =0, apenas o leopardo; valores intermediários fundem os olhares das duas imagens proporcionalmente.

3.2.3 Operações Lógicas e Máscaras Bit a Bit

As operações lógicas bit a bit (AND, OR e NOT) atuam diretamente sobre os bits de cada pixel e são a base para criação e aplicação de **máscaras** (*masks*) — imagens binárias com apenas 0 (preto) e 255 (branco) usadas para isolar Regiões de Interesse (ROI).

O comportamento de cada operação decorre da representação binária do 255 (`11111111`) e do 0 (`00000000`):

- **AND** com a máscara: onde $m = 255$, os bits originais são preservados; onde $m = 0$, o pixel é zerado. Resultado: recorte da ROI.
$$g(x, y) = f(x, y) \text{ AND } m(x, y) \tag{3.4}$$
- **OR** com a máscara: onde $m = 255$, o pixel é forçado a branco; onde $m = 0$, o valor original é mantido. Resultado: iluminação da ROI.
- **NOT** (sem máscara): inverte todos os bits ($g = 255 - f$), produzindo o negativo fotográfico da imagem.

A Figure 3.5 ilustra as três operações aplicadas à imagem do mandrill com uma máscara circular.

```
h, w = img_gray.shape

# Máscara circular centrada na imagem
mask_circ = np.zeros((h, w), dtype=np.uint8)
cv2.circle(mask_circ, (w//2, h//2), min(h, w)//3 - 10, 255, -1)

# Operações via morph
img_not = mm.bnot(img_gray)             # NOT: negativo fotográfico
```

```
img_and = mm.band(img_gray, mask_circ) # img_gray & mask_circ: preserva apenas a ROI circular
img_or  = mm.bor(img_gray, mask_circ)  # img_gray | mask_circ: ilumina a região da máscara

mm.show(
    [img_gray, img_and, img_or, img_not],
    titles=["Original", "AND (ROI circular)", "OR (ilumina ROI)", "NOT (negativo)"],
    cols=4
)
```

Figura 3.5: Operações lógicas bit a bit com máscara circular: NOT (negativo), AND (isolamento da ROI) e OR (iluminação da ROI).

3.3 Histograma de Imagens

O **histograma** de uma imagem em tons de cinza é uma função discreta que descreve a distribuição de frequências das intensidades:

$$h(r_k) = n_k, \quad k = 0, 1, \dots, L-1 \tag{3.5}$$

onde r_k é o k-ésimo nível de intensidade, n_k é o número de pixels com essa intensidade e L é o total de níveis (tipicamente 256 para 8 bits). O histograma normalizado estima a probabilidade de cada nível:

$$p(r_k) = \frac{n_k}{MN} \tag{3.6}$$

onde MN é o total de pixels. Por ser uma **estatística global**, o histograma não carrega informação posicional, mas revela características essenciais como brilho médio, contraste e distribuição tonal. Na prática, `mm.hist(img)` retorna o vetor de contagens $h(r_k)$, que serve tanto para visualização (via `mm.histImg`) quanto para cálculos como **função de distribuição acumulada (CDF)** e equalização.

i Interpretação do Histograma

- **Estreito à esquerda:** imagem subexposta (escura).
- **Estreito à direita:** imagem superexposta (clara).
- **Concentrado no centro:** baixo contraste.
- **Distribuído por toda a faixa:** alto contraste, boa utilização dos tons disponíveis.

A Figure 3.6 exibe o histograma da imagem do mandrill, de uma versão escurecida (`mm.subm`) e de uma versão clareada (`mm.addm`), evidenciando o deslocamento da distribuição para a esquerda e para a direita, respectivamente.

```
img_dark = mm.subm(img_gray, 80)
img_high = mm.addm(img_gray, 80)

images = [img_gray, img_dark, img_high]
titles = ["Original", "Escurecida (-80)", "Clareada (+80)"]

def hist_title(img, t):
```

```
    H = mm.hist(img)
    n0   = int(H[0])
    n255 = int(H[255]) if len(H) > 255 else 0
    return f"Histograma - {t}\n0:{n0:,}px  255:{n255:,}px"

mm.show(
    images + [mm.histImg(img) for img in images],
    titles=titles + [hist_title(img, t) for img, t in zip(images, titles)],
    rows=2, cols=3,
    figsize=(14, 8)
)
```

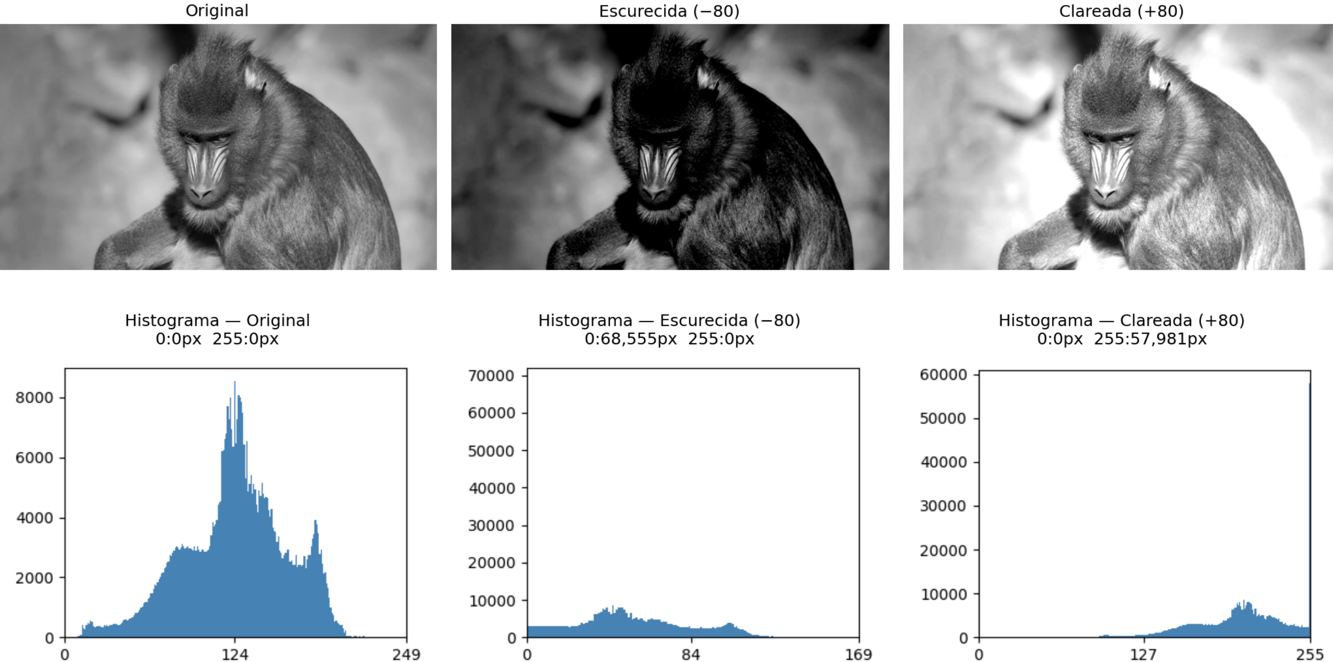


Figura 3.6: Histogramas da imagem original, de uma versão escurecida e de uma versão clareada. Os valores entre parênteses indicam a quantidade de pixels saturados em 0 (preto) e 255 (branco).

3.3.1 Equalização de Histograma

A **equalização de histograma** redistribui as intensidades para que o histograma resultante seja o mais uniforme possível. O mapeamento é dado pela **função de distribuição acumulada (CDF)**:

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p(r_j) = \frac{L-1}{MN}\sum_{j=0}^{k} n_j \tag{3.7}$$

A transformação é monotônica: níveis frequentes recebem intervalos maiores no domínio de saída (maior separação → mais contraste), enquanto níveis raros são comprimidos.

O algoritmo completo, em cinco etapas, é apresentado na Table 3.1.

Tabela 3.1: Algoritmo de equalização de histograma.

Etapa	Operação	Fórmula
1	**Histograma**	$h[k] \leftarrow$ número de pixels com intensidade k, $k = 0 \dots L-1$
2	**Probabilidade**	$p[k] \leftarrow h[k]/MN$

Etapa	Operação	Fórmula
3	**CDF**	$\text{cdf}[k] \leftarrow \sum_{j=0}^{k} p[j]$ (soma acumulada)
4	**Mapeamento (LUT)**	$\text{lut}[k] \leftarrow \text{round}(\text{cdf}[k] \times (L-1))$
5	**Aplicação**	$g[i,j] \leftarrow \text{lut}[f[i,j]]$ (para todo pixel)

Note na Figure 3.7 que a equalização **redistribui** os tons existentes para posições mais espaçadas na faixa $[0, L-1]$, mas não cria novos tons — a imagem equalizada continua com exatamente 3 tons distintos, agora em $\{1, 5, 7\}$ em vez de $\{2, 3, 4\}$.

```
img5 = np.array([[3, 4, 2, 3, 4],
                 [4, 3, 3, 4, 3],
                 [2, 3, 4, 3, 2],
                 [3, 4, 3, 2, 3],
                 [4, 3, 2, 3, 4]], dtype=np.uint8)
L = 8  # 3 bits: níveis 0..7

# 1. Histograma com tamanho garantido até L
h_raw = mm.hist(img5)
h = np.zeros(L, dtype=int)
h[:len(h_raw)] = h_raw  # h[k] = nº pixels com intensidade k

p   = h / h.sum()  # 2. Probabilidade de cada nível p[k] = h[k] / MN

cdf = np.cumsum(p) # 3. CDF normalizada ou
# cdf = np.zeros(len(p))
# cdf[0] = p[0]
# for k in range(1, len(p)):
#     cdf[k] = cdf[k-1] + p[k] # cdf[k] = Σ p[j], j=0..k

lut = np.round(cdf * (L - 1)).astype(np.uint8)  # 4. LUT: mapeamento para [0, L-1]

img5_eq = lut[img5] # 5. Aplica LUT pixel a pixel ou, equivalente a:
# l, c = img5.shape
# img5_eq = np.zeros((l, c), dtype=np.uint8)
# for i in range(l):
#     for j in range(c):
#         img5_eq[i, j] = lut[img5[i, j]]      # g[i,j] = lut[f[i,j]]

print("Imagem original (5×5, 3 bits):")
print(img5)
print()
header = f"{'k':>3} {'h[k]':>6} {'p[k]':>7} {'CDF[k]':>8} {'lut[k]':>7}"
print(header)
print("-" * len(header))
for k in range(L):
    print(f"{k:>3} {h[k]:>6} {p[k]:>7.4f} {cdf[k]:>8.4f} {lut[k]:>7}")
print()
print("Imagem equalizada (5×5):")
print(img5_eq)

# Conta tons distintos
tons_orig = len(np.unique(img5))
tons_eq   = len(np.unique(img5_eq))

mm.show(
```

```
    [img5, img5_eq, mm.histImg(img5, L-1), mm.histImg(img5_eq, L-1)],
    titles=[f"Original ({tons_orig} tons: {sorted(np.unique(img5).tolist())})",
            f"Equalizada ({tons_eq} tons: {sorted(np.unique(img5_eq).tolist())})",
            "Histograma - Original",
            "Histograma - Equalizada"],
    rows=2, cols=2, figsize=(5, 4), dpi=100
)
```

```
Imagem original (5×5, 3 bits):
[[3 4 2 3 4]
 [4 3 3 4 3]
 [2 3 4 3 2]
 [3 4 3 2 3]
 [4 3 2 3 4]]

  k   h[k]    p[k]   CDF[k]  lut[k]
-----------------------------------
  0      0  0.0000   0.0000       0
  1      0  0.0000   0.0000       0
  2      5  0.2000   0.2000       1
  3     12  0.4800   0.6800       5
  4      8  0.3200   1.0000       7
  5      0  0.0000   1.0000       7
  6      0  0.0000   1.0000       7
  7      0  0.0000   1.0000       7

Imagem equalizada (5×5):
[[5 7 1 5 7]
 [7 5 5 7 5]
 [1 5 7 5 1]
 [5 7 5 1 5]
 [7 5 1 5 7]]
```

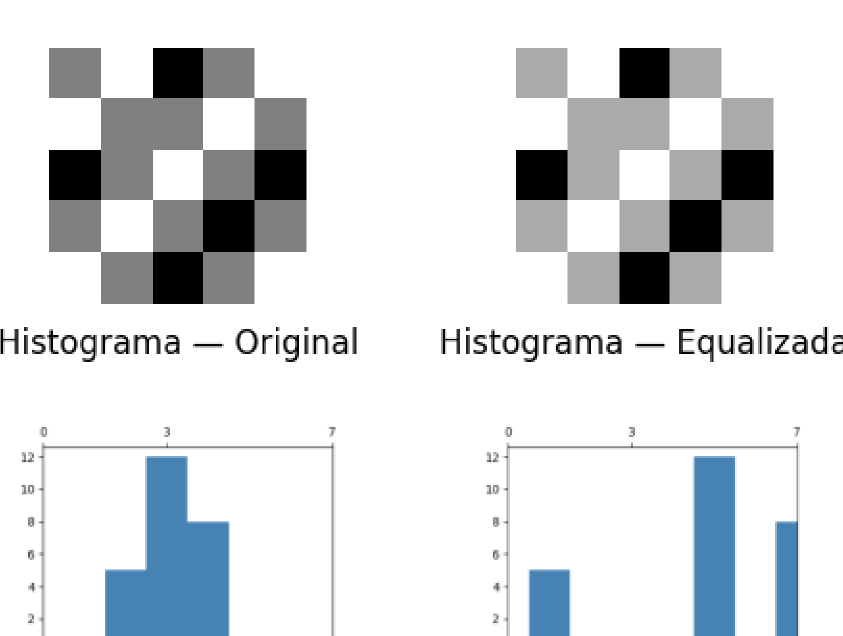


Figura 3.7: Equalização de histograma em imagem 5×5 com 3 bits (L=8): imagem original com tons concentrados em {2,3,4}, imagem equalizada com tons redistribuídos para {1,5,7}, e os respectivos histogramas evidenciando o espalhamento das frequências.

Limitação: a equalização global pode super-realçar ruídos em regiões homogêneas. O **CLAHE** (*Contrast Limited Adaptive Histogram Equalization*) resolve isso aplicando a equalização em blocos locais com um limite máximo de *clip* para o histograma de cada bloco.

A Figure 3.8 compara a imagem original, a equalização via `mm.equalize` e o CLAHE do OpenCV, exibindo também os histogramas resultantes.

```
img_eq    = mm.equalize(img_gray)
clahe     = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(32, 32))
img_clahe = clahe.apply(img_gray)

images = [img_gray, img_eq, img_clahe]
titles = ["Original", "mm.equalize (CDF)", "CLAHE"]

mm.show(
    images + [mm.histImg(img) for img in images],
    titles=titles + [f"Histograma - {t}" for t in titles],
    rows=2, cols=3,
    figsize=(14, 8)
)
```

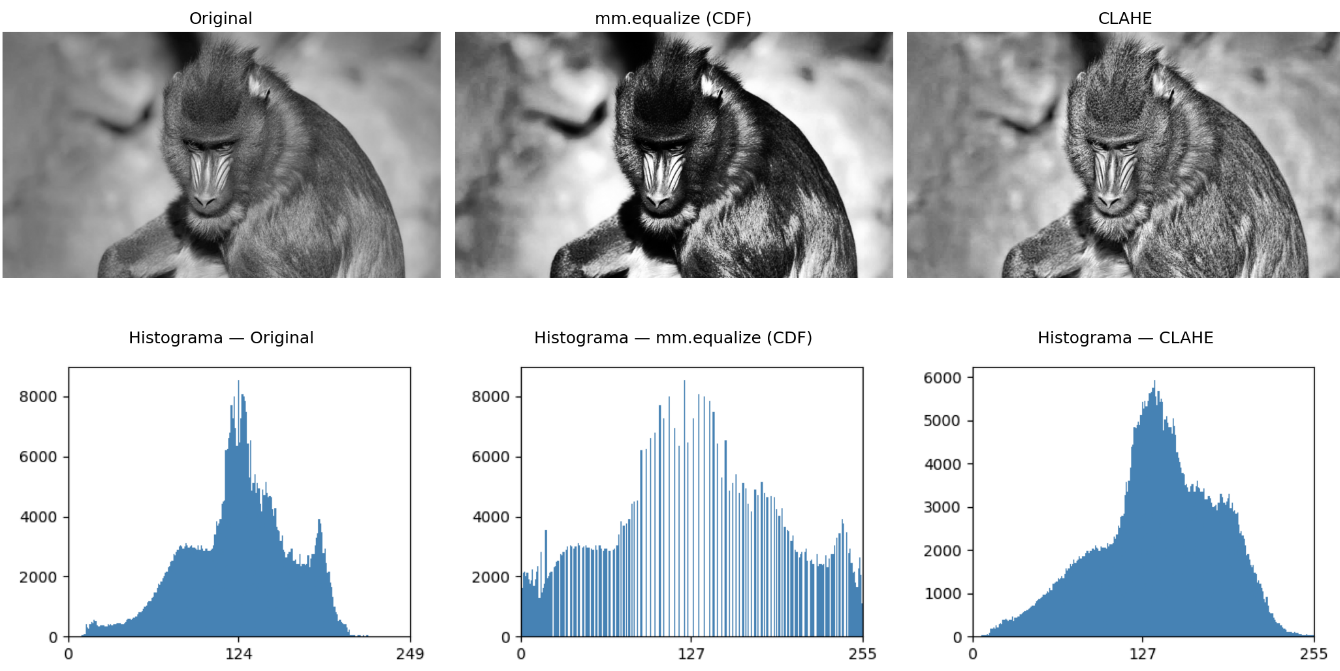


Figura 3.8: Equalização de histograma: mm.equalize (global via CDF) vs. CLAHE (adaptativa com limitação de contraste). Os histogramas revelam o espalhamento progressivo das intensidades.

3.3.2 Especificação de Histograma

Enquanto a equalização impõe uma distribuição uniforme, a **especificação de histograma** (*histogram matching*) permite que o histograma da imagem de saída siga uma distribuição **arbitrária** — por exemplo, o histograma de outra imagem de referência.

O procedimento envolve três etapas:

1. Calcular a CDF da imagem de entrada: $P_r(r_k)$.
2. Calcular a CDF da imagem de referência: $P_z(z_k)$.
3. Para cada nível r_k, encontrar o nível z que minimiza $|P_z(z) - P_r(r_k)|$.

$$T(r_k) = \arg\min_z |P_z(z) - P_r(r_k)| \tag{3.8}$$

Na Figure 3.9, transferimos o perfil tonal do leopardo (Figure 3.3) para a imagem do mandrill — uma aplicação direta do conceito visto no *blending*: em vez de fundir pixels, aqui fundimos distribuições tonais.

```
img_leop_gray = mm.gray(img_numpy)

def hist_specify(src, ref):
    """Mapeia src para o perfil tonal de ref via CDF."""
    cdf_src = np.cumsum(mm.hist(src) / src.size)
    cdf_ref = np.cumsum(mm.hist(ref) / ref.size)
    lut = np.array([np.argmin(np.abs(cdf_ref - v)) for v in cdf_src], dtype=np.uint8)
    return lut[src]

img_spec = hist_specify(img_gray, img_leop_gray)

images = [img_gray, img_leop_gray, img_spec]
titles = ["Mandrill (original)", "Leopardo (referência)", "Mandrill → perfil do leopardo"]

mm.show(
    images + [mm.histImg(img) for img in images],
```

```
    titles=titles + [f"Histograma - {t}" for t in titles],
    rows=2, cols=3,
    figsize=(12, 9)
)
```

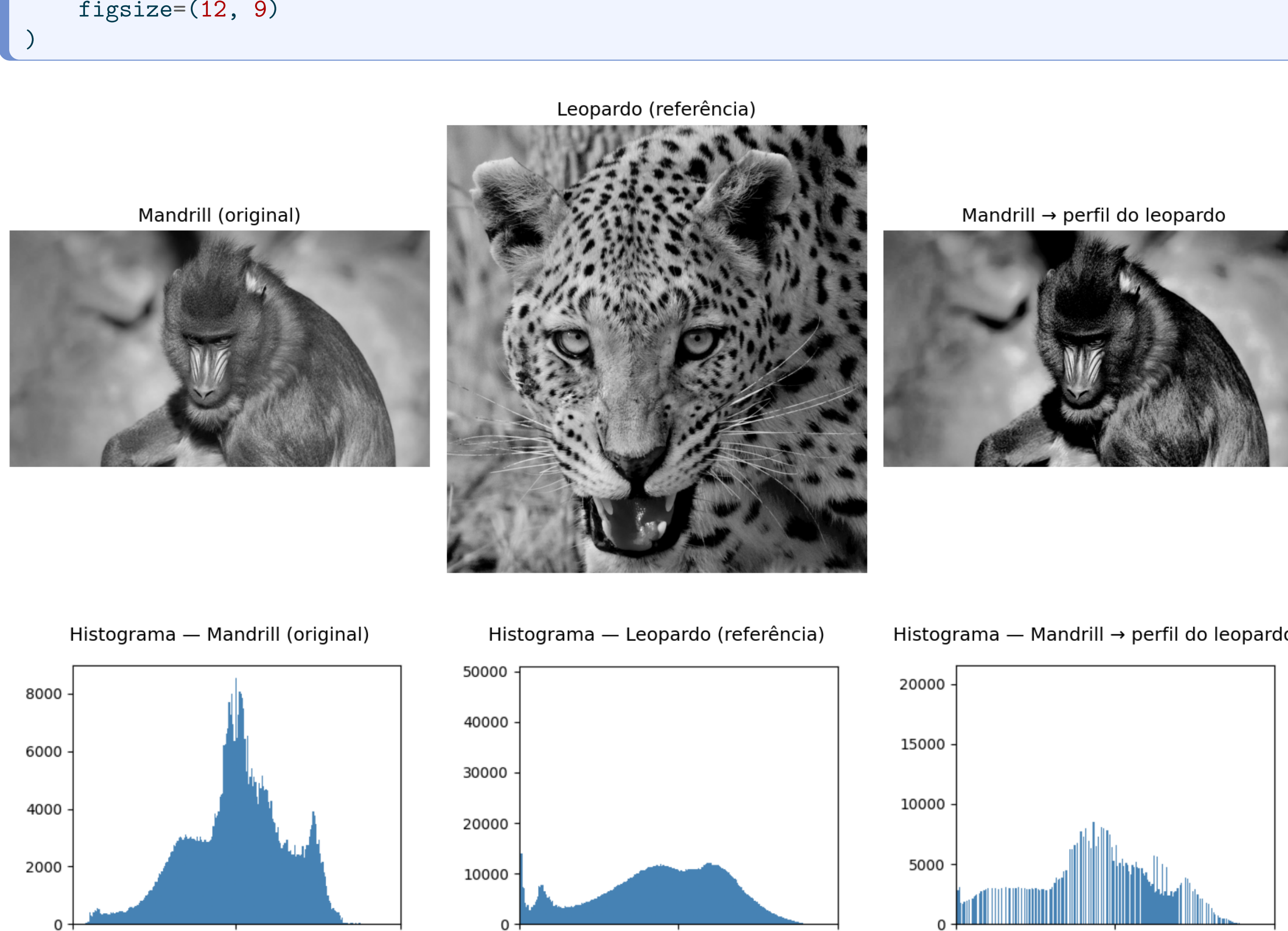


Figura 3.9: Especificação de histograma: mandril mapeada para o perfil tonal do leopardo. A CDF da saída aproxima a CDF de referência.

3.4 Fundamentos Espaciais: Vizinhança, Convolução e *Kernels*

As operações de filtragem espacial operam sobre **regiões vizinhas** de pixels — não mais pixel a pixel. O conceito central é a **janela deslizante** (*sliding window*): um *kernel* w de dimensão $(2a+1) \times (2b+1)$ percorre toda a imagem e, para cada posição (x, y), calcula uma nova intensidade combinando os pixels da vizinhança com os coeficientes do *kernel*.

3.4.1 Vizinhança e Borda

Para um pixel (x, y), a vizinhança retangular de raio (a, b) é o conjunto:

$$\mathcal{V}(x, y) = \{(x+s,\ y+t) : -a \le s \le a,\ -b \le t \le b\} \tag{3.9}$$

Pixels próximos à borda da imagem têm vizinhanças parcialmente fora do domínio. As estratégias mais comuns são: **zero-padding** (completa com zeros), **replicação** (repete o pixel de borda) e **reflexão** (espelha a imagem). O OpenCV usa replicação por padrão (`cv2.BORDER_REFLECT_101`).

3.4.2 Correlação e Convolução

Existem dois mecanismos matematicamente relacionados:

Correlação cruzada (*cross-correlation*) — o *kernel* é aplicado diretamente:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)\, f(x+s,\ y+t) \tag{3.10}$$

Convolução bidimensional — o *kernel* é rotacionado 180° antes da aplicação:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)\, f(x-s,\ y-t) \tag{3.11}$$

Para *kernels* simétricos (Gaussiano, Laplaciano, média) as duas operações produzem resultados idênticos. Para *kernels* assimétricos (Sobel, Prewitt) a diferença é significativa.

> **i** Correlação vs. Convolução no OpenCV
>
> `cv2.filter2D` implementa **correlação**. Para convolução verdadeira com *kernel* assimétrico, rotacione o *kernel* 180° (`np.rot90(w, 2)`) antes de passar para `filter2D`.

3.4.3 O Papel do *Kernel*

Os coeficientes do *kernel* determinam completamente o efeito do filtro:

Tabela 3.2: Interpretação dos coeficientes do *kernel*.

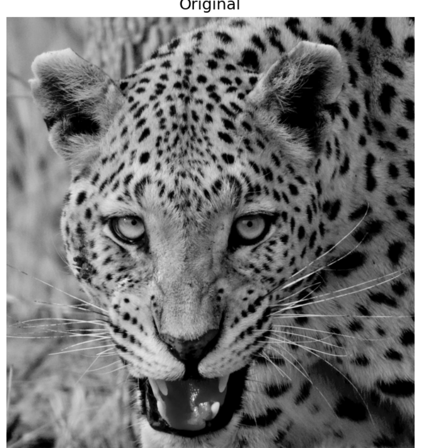
Soma	Coeficientes	Efeito
$=1$	todos positivos	Suavização — passa-baixa
$=0$	positivos e negativos	Detecção de bordas — passa-alta
$=1$	centro >1, bordas <0	Realce de nitidez
—	assimétricos	Gradiente direcional

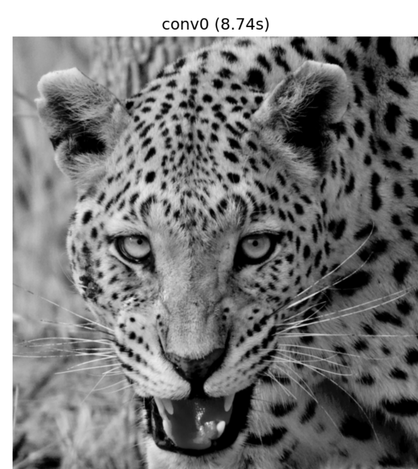
A Figure 3.10 demonstra o mecanismo passo a passo: para cada posição da janela, multiplica-se o *kernel* pela vizinhança e soma-se o resultado — exatamente a Equation 3.10 avaliada em um único ponto.

```python
import time

w_mean = np.ones((3,3), dtype=np.float32) / 9.0
img_gray = img_leop_gray
# Demonstração numérica em patch 5×5
patch = img_gray[250:255, 250:255].astype(np.float32)
roi   = patch[0:3, 0:3]
print("Patch 5×5 (intensidades):")
print(patch.astype(np.int32))
print(f"\nKernel 3×3 de média:\n{w_mean}")
print(f"\nCorrelação no pixel central [1,1]: \
      {(w_mean * roi).sum():.1f}  (original: {patch[1,1]:.0f})")

# Comparação de tempo na imagem completa (Mandrill)
t0 = time.perf_counter()
img_conv0 = mm.conv0(img_gray, w_mean)
t_conv0 = time.perf_counter() - t0

t0 = time.perf_counter()
img_conv = mm.conv(img_gray, w_mean)
t_conv = time.perf_counter() - t0
```

```
diff = np.abs(img_conv0.astype(int) - img_conv.astype(int)).max()
print(f"\nTempos na imagem {img_gray.shape} (Mandrill):")
print(f"  mm.conv0 (laços Python): {t_conv0:.3f} s")
print(f"  mm.conv  (cv2.filter2D): {t_conv:.5f} s")
print(f"  Aceleração:              {t_conv0/t_conv:.0f}× mais rápido")
print(f"  Diferença máxima:        {diff} (bordas com padding diferente)")

mm.show(
    [img_gray, img_conv0, img_conv],
    titles=["Original", f"conv0 ({t_conv0:.2f}s)", f"conv ({t_conv:.4f}s)"],
    cols=3
)
```

```
Patch 5×5 (intensidades):
[[ 95  97 103 113 115]
 [107 105 103 108 115]
 [ 98 102 112 118 116]
 [ 81  91 113 123 119]
 [ 85  91 111 120 121]]

Kernel 3×3 de média:
[[0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]]

Correlação no pixel central [1,1]:       102.4  (original: 105)

Tempos na imagem (1324, 1242) (Mandrill):
  mm.conv0 (laços Python): 8.742 s
  mm.conv  (cv2.filter2D): 0.00206 s
  Aceleração:              4247× mais rápido
  Diferença máxima:        43 (bordas com padding diferente)
```





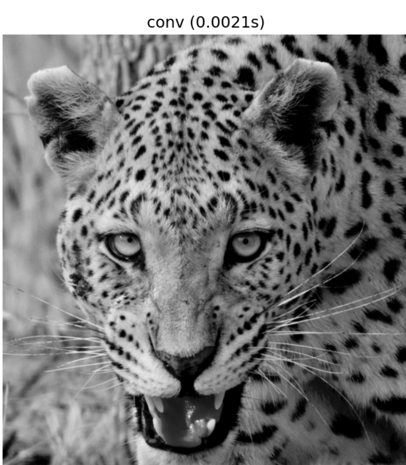


Figura 3.10: Correlação com kernel de média 3×3: versão didática (mm.conv0) vs. vetorizada (mm.conv via cv2.filter2D). A diferença máxima de 27 ocorre nas bordas, onde o tratamento de padding difere.

3.4.4 Exemplo Numérico: Correlação Passo a Passo

Para tornar concreto o mecanismo da Equation 3.10, considere o *kernel* de média 3×3 ($a = b = 1$, todos os coeficientes $= 1/9 \approx 0{,}111$) aplicado ao patch 5×5 extraído da imagem do mandrill. A Figure 3.11 exibe o patch com grade e destaca em amarelo a janela 3×3 centrada no pixel $[1, 1]$:

```
patch = img_gray[250:255, 250:255]
B     = np.ones((3,3), dtype=np.uint8)

print("Patch 5×5 (intensidades):")
print(mm.drawImage(patch))

mm.drawImageKernel(patch, B, x=1, y=1, scale=40.0)
```

```
Patch 5×5 (intensidades):
 95  97 103 113 115
107 105 103 108 115
 98 102 112 118 116
 81  91 113 123 119
 85  91 111 120 121
```

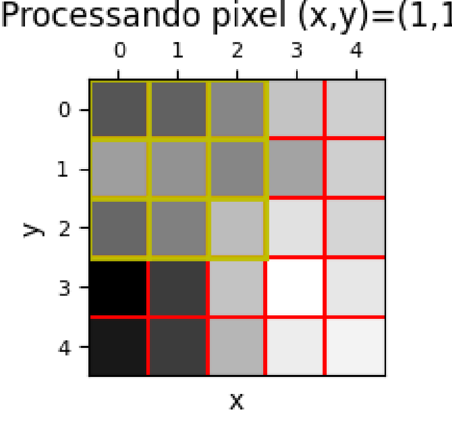


Figura 3.11: Patch 5×5 extraído da imagem do mandrill (posição [250:255, 250:255]). A janela amarela destaca a vizinhança 3×3 centrada no pixel [1,1] onde a correlação será calculada.

Os valores da vizinhança 3×3 centrada em $[1, 1]$ são exatamente a submatriz superior esquerda do patch:

$$\text{vizinhança} = \begin{bmatrix} 187 & 189 & 192 \\ 190 & 196 & 197 \\ 193 & 197 & 199 \end{bmatrix}$$

Aplicando a Equation 3.10 com o kernel de média:

$$g[1, 1] = \frac{187 + 189 + 192 + 190 + 196 + 197 + 193 + 197 + 199}{9} = \frac{1740}{9} \approx 193$$

O resultado (193) é ligeiramente menor que o original (196) porque a média inclui vizinhos de menor intensidade — efeito típico de suavização. A janela desliza então para o próximo pixel $[1, 2]$, recalcula com uma nova vizinhança 3×3, e assim sucessivamente até cobrir toda a imagem.

⚠ Desempenho: laços Python vs. operações vetorizadas

A função `mm.conv0` realiza a convolução com laços Python explícitos, processando pixel a pixel. Na imagem Mandrill (540×960), isso levou cerca de 3 segundos para um kernel 3×3.
Já `mm.conv` utiliza `cv2.filter2D`, implementado em C++ com operações vetorizadas e otimizações internas, executando a mesma operação em apenas 0.00068 s — aproximadamente **4448× mais rápido**:

```
mm.conv0 (laços Python): 3.005 s
mm.conv  (cv2.filter2D): 0.00068 s
Aceleração:              4448× mais rápido
```

As pequenas diferenças entre os resultados (diferença máxima igual a 27) decorrem principalmente do tratamento distinto das bordas (*padding*).
Por isso, `mm.conv0` deve ser usado apenas para fins didáticos. Em aplicações reais, utilize sempre implementações vetorizadas como `mm.conv`.

3.5 Filtragem Espacial de Suavização

Os filtros de suavização (*smoothing filters*) atenuam variações bruscas de intensidade, reduzindo ruído e detalhes de alta frequência. São filtros **passa-baixa** — preservam as componentes de baixa frequência (estruturas grandes) e atenuam as de alta frequência (ruído, bordas).

3.5.1 Filtro de Média (*Box Filter*)

O filtro de média utiliza um *kernel* uniforme de tamanho $n \times n$, onde todos os coeficientes valem $1/n^2$:

$$w_{\text{média}} = \frac{1}{n^2} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}_{n \times n} \tag{3.12}$$

Cada pixel de saída é a média aritmética dos n^2 pixels de sua vizinhança. Note que a soma dos coeficientes é sempre 1 — o brilho médio da imagem é preservado. Kernels maiores produzem suavização mais agressiva, mas borram progressivamente as bordas.

A Figure 3.12 compara o efeito de kernels de média com tamanhos crescentes na imagem do mandrill completa. Observe como o borramento das bordas aumenta proporcionalmente ao tamanho do kernel.

```python
# Detalhe da região do olho
y0, y1, x0, x1 = 580, 740, 680, 900
crop = lambda img: img[y0:y1, x0:x1]

img_gray_crop = crop(img_gray)

sizes = [3, 7, 15]
imgs  = [img_gray_crop] + \
    [mm.conv(img_gray_crop, np.ones((k,k), dtype=np.float32)/(k*k)) for k in sizes]
titles = ["Original"] + [f"Média {k}×{k}" for k in sizes]

mm.show(imgs, titles=titles, cols=4)
```

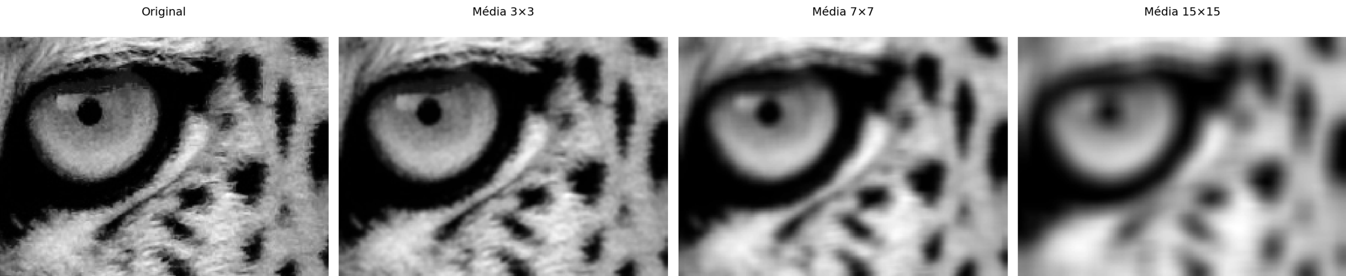


Figura 3.12: Filtro de média com kernels de tamanho crescente (3×3, 7×7, 15×15). O borramento das bordas aumenta com o tamanho do kernel.

3.5.2 Filtro Gaussiano

O filtro Gaussiano pesa os pixels da vizinhança de acordo com uma função Gaussiana bidimensional:

$$G(s,t) = \frac{1}{2\pi\sigma^2}\, e^{-\frac{s^2+t^2}{2\sigma^2}} \tag{3.13}$$

onde σ é o desvio padrão e controla o raio de influência. Pixels mais próximos do centro têm peso maior; pixels distantes são progressivamente ignorados.

Para visualizar o kernel antes de aplicá-lo, a Figure 3.13 exibe o kernel Gaussiano 5×5 gerado para $\sigma = 1$ — note a queda radial dos pesos a partir do centro:

```
# Gera kernel Gaussiano 5×5 via cv2
k_gauss = cv2.getGaussianKernel(5, 1)
w_gauss = k_gauss @ k_gauss.T          # produto externo: G(s,t) = G(s)·G(t)
w_gauss = w_gauss / w_gauss.sum()      # garante soma = 1

print("Kernel Gaussiano 5×5 ( =1), normalizado:")
for row in w_gauss:
    print("  " + "  ".join(f"{v:.4f}" for v in row))
print(f"\nSoma dos coeficientes: {w_gauss.sum():.6f}")
print(f"Peso central vs. canto: {w_gauss[2,2]:.4f} vs. {w_gauss[0,0]:.4f} "
      f"({w_gauss[2,2]/w_gauss[0,0]:.1f}× maior)")

mm.drawImagePlt((w_gauss * 1000).astype(np.uint8), scale=40)
```

```
Kernel Gaussiano 5×5 ( =1), normalizado:
  0.0030  0.0133  0.0219  0.0133  0.0030
  0.0133  0.0596  0.0983  0.0596  0.0133
  0.0219  0.0983  0.1621  0.0983  0.0219
  0.0133  0.0596  0.0983  0.0596  0.0133
  0.0030  0.0133  0.0219  0.0133  0.0030

Soma dos coeficientes: 1.000000
Peso central vs. canto: 0.1621 vs. 0.0030 (54.6× maior)
```

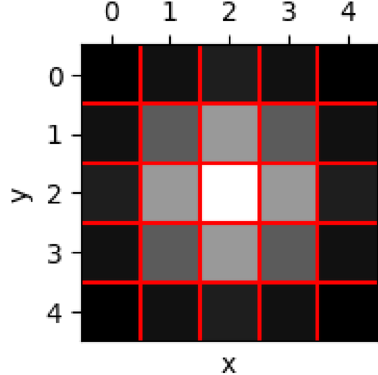


Figura 3.13: Kernel Gaussiano 5×5 (=1): pesos normalizados, maiores no centro e decrescentes radialmente. A grade facilita a leitura de cada coeficiente.

A propriedade de **separabilidade** — $G(s,t) = G(s) \cdot G(t)$ — é computacionalmente valiosa: em vez de uma convolução 2D com n^2 operações por pixel, aplica-se duas convoluções 1D com $2n$ operações, reduzindo a complexidade de $O(n^2)$ para $O(n)$.

Comparado ao filtro de média, o Gaussiano:

- **Preserva melhor as bordas** — a ponderação radial suaviza sem criar transições abruptas;
- **Não introduz anéis** (*ringing*) no domínio da frequência, pois a Gaussiana é sua própria transformada de Fourier;
- **É controlado por σ** — aumentar σ equivale a aumentar o raio de suavização de forma contínua e previsível.

A Figure 3.14 compara o filtro de média e o Gaussiano aplicados à imagem do mandrill com janela 9×9:

```
w_media9 = np.ones((9,9), dtype=np.float32) / 81.0
img_media9 = mm.conv(img_gray, w_media9)
img_gauss9 = cv2.GaussianBlur(img_gray, (9,9), 0)

# Detalhe da região do olho
y0, y1, x0, x1 = 580, 740, 680, 900
crop = lambda img: img[y0:y1, x0:x1]

img_gray_crop = crop(img_gray)


mm.show(
    [crop(img_gray),  crop(img_media9),  crop(img_gauss9)],
    titles=["Detalhe: Original", "Média 9×9", "Gaussiano 9×9"],
    cols=3, figsize=(12, 8)
)
```

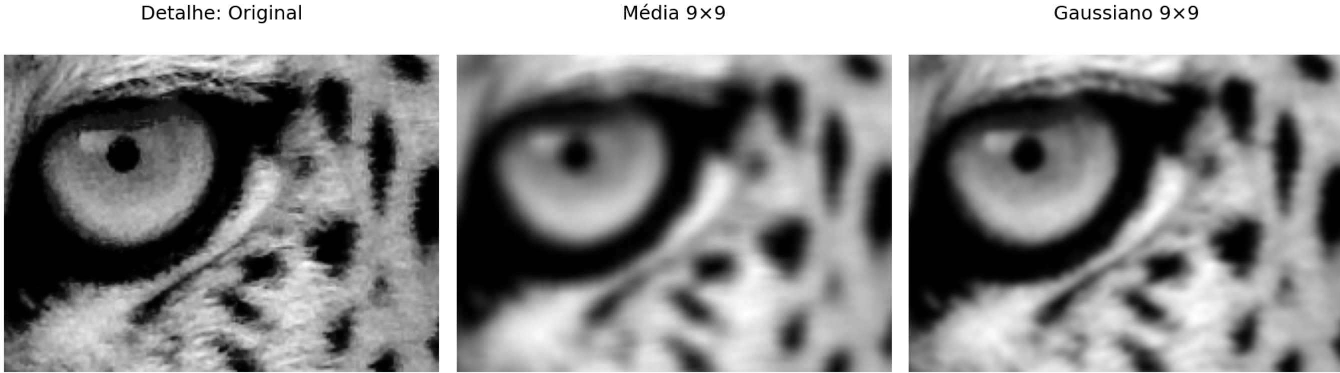


Figura 3.14: Comparação entre filtro de média e Gaussiano (janela 9×9, =0 para cálculo automático). O Gaussiano preserva melhor as bordas, visível no detalhe do rosto.

3.6 Filtragem Espacial de Realce

Os filtros de realce (*sharpening filters*) enfatizam transições abruptas de intensidade, aumentando a nitidez e a visibilidade de bordas. São filtros **passa-alta** — amplificam as componentes de alta frequência (bordas, textura) e suprimem as de baixa frequência (regiões uniformes).

A intuição é simples: se subtrairmos de uma imagem sua versão suavizada (que contém apenas as baixas frequências), o que resta são as altas frequências — bordas e detalhes. Somando esse resíduo de volta à imagem original, o contraste local aumenta:

$$g = f + k\,(f - f_{\text{suave}}), \quad k > 0 \tag{3.14}$$

Os filtros de realce formalizam essa ideia diretamente no *kernel*, sem precisar de duas etapas separadas.

3.6.1 Laplaciano

O Laplaciano é um operador de **segunda derivada** isotrópico — responde igualmente a variações em todas as direções, ao contrário dos operadores de primeira derivada (Sobel, Prewitt) que são direcionais:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{3.15}$$

A segunda derivada tem uma propriedade fundamental para o realce: vale **zero em regiões uniformes**, é **positiva antes de uma borda** (intensidade crescente) e **negativa após** (intensidade decrescente). Subtrair o Laplaciano da imagem original, portanto, aumenta o contraste exatamente onde existem transições:

$$g(x,y) = f(x,y) - \nabla^2 f(x,y) \tag{3.16}$$

Na forma discreta, a segunda derivada em x é aproximada por $f(x+1,y)-2f(x,y)+f(x-1,y)$, e analogamente em y. Somando as duas direções, obtém-se o kernel w_4 (4-vizinhos) ou w_8 (8-vizinhos, incluindo diagonais):

$$w_4 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \qquad w_8 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.17}$$

i Soma zero e centro negativo

Ambos os kernels têm **soma dos coeficientes igual a zero**: em regiões uniformes, a saída é 0 — o Laplaciano não altera o brilho médio, apenas detecta variações. O **centro negativo** indica que o pixel é comparado com seus vizinhos: quanto mais ele se destacar (para cima ou para baixo), maior o valor absoluto do Laplaciano naquele ponto.

No exemplo, o pixel central $[1,1] = 196$ tem vizinhos $\{189, 190, 197, 197\}$. O Laplaciano retorna um valor próximo de zero porque a região é quase uniforme — o realce será mínimo. Em regiões de borda, onde os vizinhos diferem muito do centro, o Laplaciano retorna valores altos (positivos ou negativos), e a subtração em Equation 3.16 amplifica a transição.

A seguir aplica os dois *kernels* à imagem do mandrill completa, comparando a resposta bruta do Laplaciano com a imagem realçada:

No exemplo, o pixel $[1,1] = 196$ está em região quase uniforme — o Laplaciano retorna valor próximo de zero e o realce é mínimo. Em pixels de borda, onde os vizinhos diferem muito do centro, o Laplaciano retorna valores altos e a Equation 3.16 amplifica a transição significativamente.

A Figure 3.16 aplica os dois kernels à imagem do mandrill completa, comparando a resposta bruta com a imagem realçada:

```
w4 = np.array([[0, 1, 0],
               [1,-4, 1],
               [0, 1, 0]], dtype=np.float32)
w8 = np.array([[1, 1, 1],
               [1,-8, 1],
               [1, 1, 1]], dtype=np.float32)

def apply_lap(img, w):
    lap     = mm.conv(img, w).astype(np.float32) - 128
    lap_viz = np.clip(np.abs(lap) / np.abs(lap).max() * 255, 0, 255).astype(np.uint8)
    realce  = np.clip(img.astype(np.float32) - lap, 0, 255).astype(np.uint8)
    return lap_viz, realce
```

```
patch  = img_gray[250:255, 250:255]
w4     = np.array([[0, 1, 0],
                   [1,-4, 1],
                   [0, 1, 0]], dtype=np.float32)
B      = np.ones((3,3), dtype=np.uint8)

p      = patch.astype(np.float32)
lap_11 = int(p[0,1] + p[1,0] - 4*p[1,1] + p[1,2] + p[2,1])
real_11 = int(np.clip(p[1,1] - lap_11, 0, 255))

print("Patch original:")
print(mm.drawImage(patch))
print(f"Laplac. w4 em  [1,1]: \
      {p[0,1]:.0f} + {p[1,0]:.0f} - 4×{p[1,1]:.0f} + {p[1,2]:.0f} + {p[2,1]:.0f} = {lap_11}")
print(f"Pixel realçado [1,1]: {p[1,1]:.0f} - ({lap_11}) = {real_11}")

# mm.drawImageKernel(patch, B, x=1, y=1, scale=20.0)
```

```
Patch original:
 95  97 103 113 115
107 105 103 108 115
 98 102 112 118 116
 81  91 113 123 119
 85  91 111 120 121

Laplac. w4 em  [1,1]:       97 + 107 - 4×105 + 103 + 102 = -11
Pixel realçado [1,1]: 105 - (-11) = 116
```

Figura 3.15

```
lap4_viz, realce4 = apply_lap(img_gray_crop, w4)
lap8_viz, realce8 = apply_lap(img_gray_crop, w8)

mm.show(
    [img_gray_crop, lap4_viz,       realce4,
     img_gray_crop, lap8_viz,       realce8],
    titles=["Original",  "Laplaciano w4",  "Realce w4",
            "Original",  "Laplaciano w8",  "Realce w8"],
    rows=2, cols=3, figsize=(14, 8)
)
```

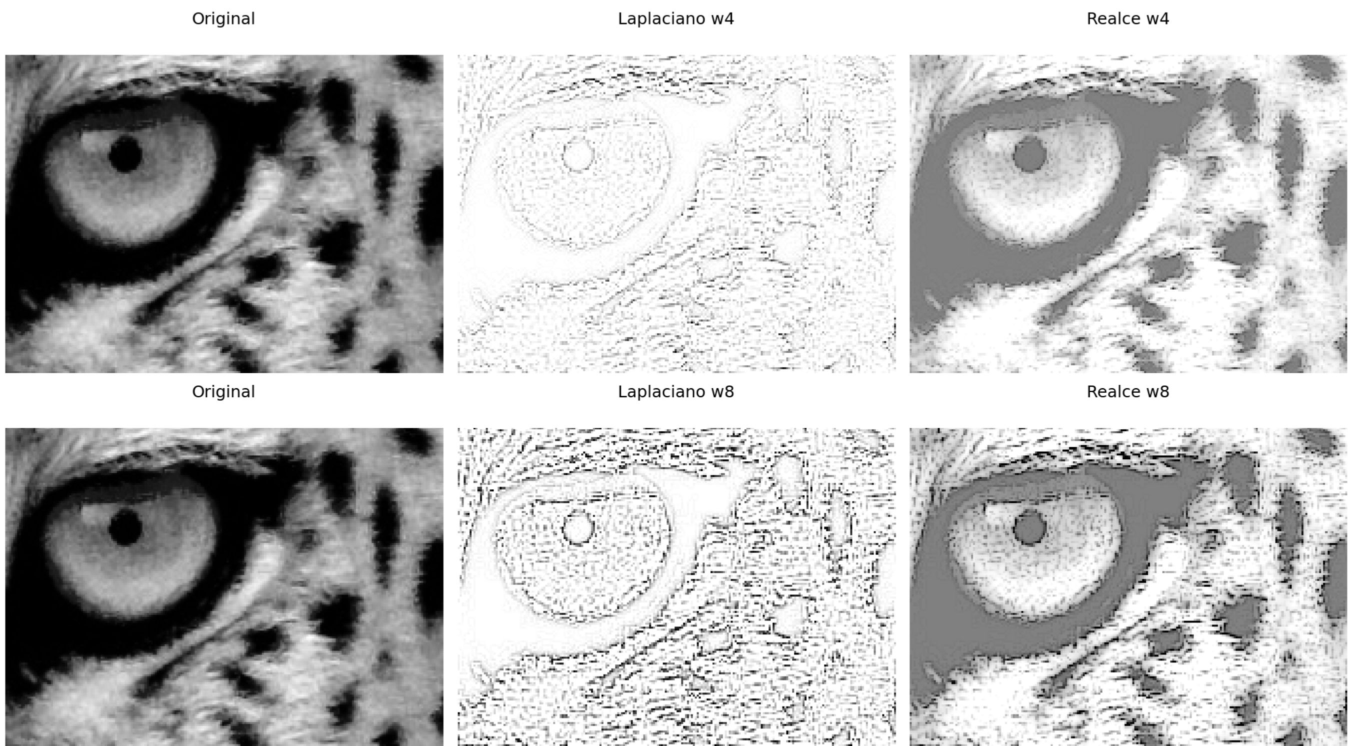


Figura 3.16: Laplaciano aplicado à imagem do mandrill: resposta bruta (bordas detectadas) com w4 e w8, e imagens realçadas pela subtração do Laplaciano. w8 é mais sensível às diagonais.

3.6.2 Operador de Sobel

O operador de Sobel estima as **derivadas parciais de primeira ordem** nas direções horizontal e vertical. Diferente do Laplaciano (segunda derivada, isotrópico), o Sobel é direcional e mais robusto ao ruído, pois cada kernel combina uma derivada com uma suavização Gaussiana perpendicular:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * f, \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * f \tag{3.18}$$

G_x detecta bordas **verticais** (variação na direção x); G_y detecta bordas **horizontais** (variação na direção y). Os pesos $\{1, 2, 1\}$ na direção perpendicular correspondem à suavização Gaussiana 1D, que reduz a sensibilidade ao ruído.

> **Sobel é correlação, não convolução**
>
> Os kernels de Sobel são **assimétricos** — a rotação de 180° altera o resultado. `cv2.Sobel` implementa correlação cruzada (como `cv2.filter2D`). Para obter a derivada direcional correta, os sinais já estão definidos para correlação: G_x retorna valores positivos onde a intensidade cresce da esquerda para a direita.

A magnitude do **gradiente** combina os dois componentes, representando a força da borda independente de direção:

$$|\nabla f| = \sqrt{G_x^2 + G_y^2} \tag{3.19}$$

E a **direção** do gradiente (perpendicular à borda) é:

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \tag{3.20}$$

Para ilustrar numericamente, calculamos G_x e G_y manualmente no pixel central $[1, 1]$ do patch 5×5:

```
patch = img_gray[250:255, 250:255]
p     = patch.astype(np.float32)

# Gx no pixel [1,1]: coluna direita - coluna esquerda (ponderada)
gx = (p[0,2] + 2*p[1,2] + p[2,2]) - (p[0,0] + 2*p[1,0] + p[2,0])

# Gy no pixel [1,1]: linha inferior - linha superior (ponderada)
gy = (p[2,0] + 2*p[2,1] + p[2,2]) - (p[0,0] + 2*p[0,1] + p[0,2])

mag   = np.sqrt(gx**2 + gy**2)
theta = np.degrees(np.arctan2(gy, gx))

print("Patch 5x5:")
print(mm.drawImage(patch))
print(f"Gx em [1,1]: ({p[0,2]:.0f} + 2*{p[1,2]:.0f} + {p[2,2]:.0f}) \
      - ({p[0,0]:.0f} + 2*{p[1,0]:.0f} + {p[2,0]:.0f}) = {gx:.1f}")
print(f"Gy em [1,1]: ({p[2,0]:.0f} + 2*{p[2,1]:.0f} + {p[2,2]:.0f}) \
      - ({p[0,0]:.0f} + 2*{p[0,1]:.0f} + {p[0,2]:.0f}) = {gy:.1f}")

# Substituído o | pelo prefixo mag para não quebrar o interpretador de tabelas do Quarto
print(f"mag(Grad f)  = sqrt({gx:.1f}^2 + {gy:.1f}^2) = {mag:.1f}")
print(f"Theta        = arctan({gy:.1f}/{gx:.1f}) = {theta:.1f} graus")
```

```
Patch 5x5:
 95  97 103 113 115
107 105 103 108 115
 98 102 112 118 116
 81  91 113 123 119
 85  91 111 120 121

Gx em [1,1]: (103 + 2*103 + 112)       - (95 + 2*107 + 98) = 14.0
Gy em [1,1]: (98 + 2*102 + 112)       - (95 + 2*97 + 103) = 22.0
mag(Grad f)  = sqrt(14.0^2 + 22.0^2) = 26.1
Theta        = arctan(22.0/14.0) = 57.5 graus
```

O valor pequeno de $|\nabla f|$ nesse patch confirma que a região é quase uniforme — o gradiente é alto apenas nas bordas reais da imagem. A Figure 3.17 aplica o operador à Mandrill completa, mostrando G_x, G_y e a magnitude:

```
def norm255(img):
    mn, mx = img.min(), img.max()
    return ((img - mn) / (mx - mn + 1e-9) * 255).astype(np.uint8)

sobelx    = cv2.Sobel(img_gray_crop, cv2.CV_64F, 1, 0, ksize=3)
sobely    = cv2.Sobel(img_gray_crop, cv2.CV_64F, 0, 1, ksize=3)
magnitude = np.sqrt(sobelx**2 + sobely**2)
direcao   = np.degrees(np.arctan2(sobely, sobelx))

mm.show(
    [img_gray_crop, norm255(np.abs(sobelx)), norm255(np.abs(sobely)),
     norm255(magnitude), norm255(direcao % 180)],
    titles=["Original", "Gx (bordas vert.)", "Gy (bordas horiz.)",
            "Magnitude |f|", "Direção  (mod 180°)"],
    cols=5
```

```
)
```

Figura 3.17: Operador de Sobel na imagem mandrill: Gx detecta bordas verticais, Gy detecta bordas horizontais, e a magnitude | f| combina ambos, revelando todas as bordas independente de direção.

3.6.3 *Unsharp Masking* (USM)

O *Unsharp Masking* é uma técnica clássica de realce de nitidez originária da fotografia analógica, hoje amplamente usada em software de edição de imagens. A ideia central é extrair as **componentes de alta frequência** da imagem (bordas e detalhes) e somá-las de volta à original com um peso k:

Tabela 3.3: Etapas do *Unsharp Masking*.

Etapa	Operação	Descrição
1	$\bar{f} = f * G_\sigma$	Suaviza com Gaussiana — retém baixas frequências
2	$m = f - \bar{f}$	Máscara: diferença = altas frequências (bordas)
3	$g = f + k \cdot m$	Soma ponderada da máscara à original

Substituindo a etapa 2 na etapa 3, obtém-se a expressão compacta:

$$g = f + k\,(f - f * G_\sigma) = (1 + k)\,f - k\,(f * G_\sigma) \tag{3.21}$$

O parâmetro k controla a intensidade do realce:

- $k = 0$: sem realce ($g = f$);
- $k = 1$: USM clássico — duplica a contribuição das altas frequências;
- $k > 1$: *High Boost Filtering* — amplificação além do dobro, útil para imagens muito borradas.

⚠ Amplificação de ruído

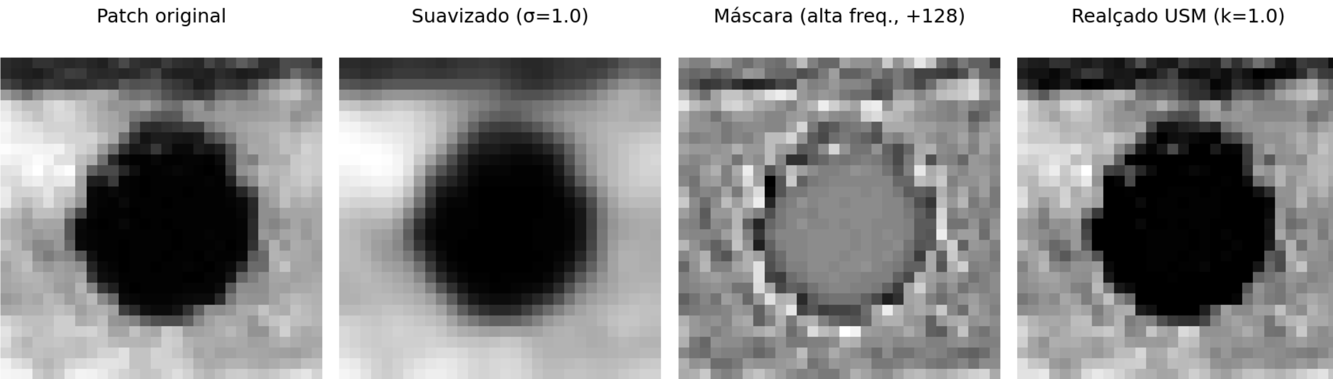
O USM não distingue bordas de ruído — ambos são componentes de alta frequência. Para k elevado, o ruído presente na imagem é amplificado junto com as bordas. Por isso, é recomendável aplicar uma leve suavização antes do USM em imagens ruidosas, ou usar σ pequeno na Gaussiana.

Para ilustrar as três etapas, aplicamos o USM ao patch 30×30 com $\sigma = 1$ e $k = 1$:

```
patch   = img_gray_crop[35:65, 45:75]
p       = patch.astype(np.float32)
sigma, k = 1.0, 1.0

# Etapa 1: suavização Gaussiana
ksize   = int(6 * sigma + 1) | 1
p_suave = cv2.GaussianBlur(p, (ksize, ksize), sigma)

# Etapa 2: máscara de alta frequência
```

```
mascara = p - p_suave

# Etapa 3: realce
p_usm = np.clip(p + k * mascara, 0, 255).astype(np.uint8)

print(f"Pixel central [1,1]: original={int(p[1,1])} suave={p_suave[1,1]:.1f}"
      f" mascara={mascara[1,1]:.1f} realcado={p_usm[1,1]}")

mm.show(
    [patch,
     p_suave.astype(np.uint8),
     np.clip(mascara + 128, 0, 255).astype(np.uint8),
     p_usm],
    titles=["Patch original",
            f"Suavizado ( ={sigma})",
            "Máscara (alta freq., +128)",
            f"Realçado USM (k={k})"],
    cols=4, figsize=(12, 4)
)
```

```
Pixel central [1,1]: original=15 suave=20.2 mascara=-5.2 realcado=9
```



A Figure 3.18 aplica o USM à imagem do mandrill completa com diferentes valores de k, evidenciando a progressão do realce:

```
def usm(img, sigma=1.0, k=1.0):
    ksize  = int(6 * sigma + 1) | 1
    suave  = cv2.GaussianBlur(img.astype(np.float32), (ksize, ksize), sigma)
    return np.clip(img.astype(np.float32) + k * (img.astype(np.float32) - suave),
                   0, 255).astype(np.uint8)

ks     = [0.5, 1.0, 2.0, 3.0]
imgs   = [img_gray_crop] + [usm(img_gray_crop, sigma=1.0, k=k) for k in ks]
titles = ["Original"] + [f"USM k={k}" for k in ks]

mm.show(imgs, titles=titles, cols=5)
```

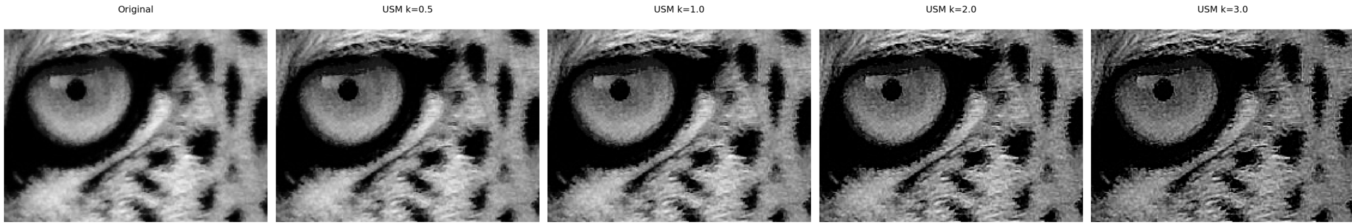


Figura 3.18: Unsharp Masking na imagem do mandrill com =1 e k variando de 0.5 a 3.0. Para k>2 surgem artefatos nas bordas (halos) e o ruído de fundo começa a ser visível.

3.7 Filtros de Ordem: Filtro da Mediana

Os filtros de ordem (*order-statistic filters*) substituem o pixel central pelo valor de um **percentil** da distribuição de intensidades da vizinhança — ao contrário dos filtros lineares, que calculam combinações ponderadas. O mais importante é o **filtro da mediana**.

3.7.1 Ruído Impulsivo: Sal e Pimenta

O ruído **sal e pimenta** (*salt-and-pepper noise*) substitui pixels aleatórios por valores extremos: 0 (pimenta, preto) ou 255 (sal, branco). É comum em transmissão de imagens com erros de bit e em câmeras com sensores defeituosos.

Para entender por que os filtros lineares falham, considere uma vizinhança 3×3 onde um único pixel foi corrompido para 255:

$$\text{vizinhança} = \begin{bmatrix} 102 & 98 & 105 \\ 100 & \mathbf{255} & 97 \\ 103 & 99 & 101 \end{bmatrix}$$

Tabela 3.4: Média vs. mediana com um pixel corrompido. A mediana ignora o outlier; a média é deslocada ~40 níveis.

Método	Cálculo	Resultado
Média	$(102 + 98 + ... + 255 + ... + 101)/9$	≈ 140
Mediana	$\{97, 98, 99, 100, \mathbf{101}, 102, 103, 105, 255\}$	101

⚠ Por que filtros de média falham com ruído impulsivo?

A média é sensível a **outliers** — um único pixel com valor 255 em uma vizinhança de valor 100 eleva a saída para 140, espalhando o ruído pela imagem. A mediana, por ser um **estimador robusto**, seleciona o valor central da distribuição ordenada, descartando naturalmente os extremos sem nenhum ajuste especial.

O experimento na Figure 3.19 ilustra a adição de ruído com diferentes densidades e a degradação progressiva da imagem:

```python
# Exemplo numérico: média vs. mediana com pixel corrompido
vizinhanca = np.array([102, 98, 105, 100, 255, 97, 103, 99, 101])
print(f"Vizinhança: {sorted([int(x) for x in vizinhanca])}")
print(f"Média:      {vizinhanca.mean():.1f}  (deslocada pelo outlier 255)")
print(f"Mediana:    {int(np.median(vizinhanca))}    (ignora o outlier)")
print(f"Valor original:  ~100")
```

```
Vizinhança: [97, 98, 99, 100, 101, 102, 103, 105, 255]
Média:      117.8  (deslocada pelo outlier 255)
Mediana:    101    (ignora o outlier)
Valor original:  ~100
```

```python
def add_salt_pepper(img, prob=0.05, seed=42):
    """Adiciona ruído sal e pimenta com probabilidade total prob."""
    noisy = img.copy()
    rnd   = np.random.default_rng(seed).random(img.shape)
    noisy[rnd < prob / 2]     = 0    # pimenta
    noisy[rnd > 1 - prob / 2] = 255  # sal
    n_corrompidos = int((rnd < prob/2).sum() + (rnd > 1-prob/2).sum())
```

```
    return noisy, n_corrompidos

probs = [0.02, 0.05, 0.10]
imgs_noise, titles_noise = [img_gray_crop], ["Original"]

for p in probs:
    noisy, n = add_salt_pepper(img_gray_crop, p)
    imgs_noise.append(noisy)
    titles_noise.append(f"Ruído {int(p*100)}%\n({n:,} pixels)")

mm.show(imgs_noise, titles=titles_noise, cols=4)
```

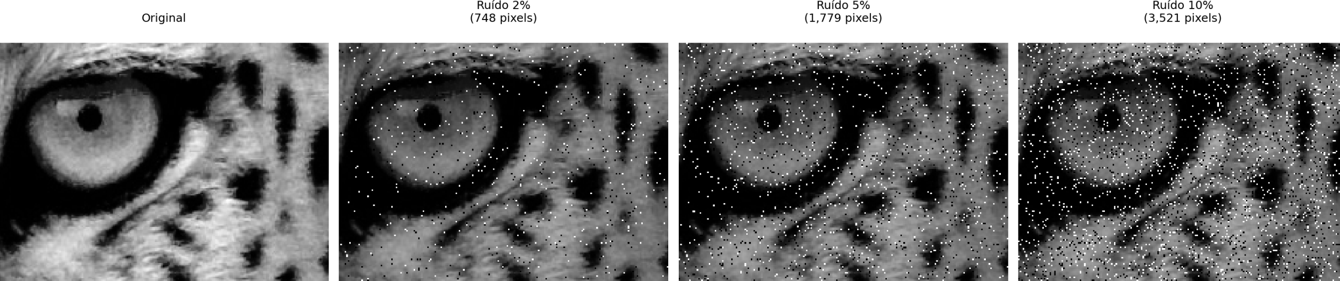


Figura 3.19: Ruído sal e pimenta com densidades crescentes (2%, 5%, 10%). O parâmetro prob indica a fração total de pixels corrompidos, metade sal (255) e metade pimenta (0).

3.7.2 Filtro da Mediana

O filtro da mediana substitui cada pixel pelo **valor mediano** dos pixels da sua vizinhança $n \times n$:

$$g(x, y) = \text{med}_{(s,t)\in\mathcal{V}_n}\{f(x + s, y + t)\} \tag{3.22}$$

O valor mediano é aquele que ocupa a posição central quando os n^2 valores da vizinhança são ordenados. Para uma janela 3×3 ($n^2 = 9$ pixels), a mediana é o 5º valor da sequência ordenada.

Para ilustrar, considere o mesmo patch 5×5 com um pixel corrompido artificialmente em $[1, 1]$:

```
patch     = img_gray[250:255, 250:255].copy()
corrompido = patch.copy()
corrompido[1, 1] = 255   # injeta pixel sal no centro

# Vizinhança 3×3 centrada em [1,1]
viz_orig = sorted(patch[0:3, 0:3].ravel().tolist())
viz_corr = sorted(corrompido[0:3, 0:3].ravel().tolist())

print("Patch original:")
print(mm.drawImage(patch))
print("\nPatch com pixel corrompido [1,1]=255:")
print(mm.drawImage(corrompido))
print(f"\nVizinhança 3×3 original (ordenada):   {viz_orig}")
print(f"Mediana original:                      {viz_orig[4]}")
print(f"\nVizinhança 3×3 corrompida (ordenada): {viz_corr}")
print(f"Mediana corrompida:                    {viz_corr[4]}  ← ignora o 255")
print(f"Média  corrompida:                 {np.mean(viz_corr):.1f}  ← distorcida pelo 255")
```

```
Patch original:
 95  97 103 113 115
107 105 103 108 115
 98 102 112 118 116
```

```
 81  91 113 123 119
 85  91 111 120 121


Patch com pixel corrompido [1,1]=255:
 95  97 103 113 115
107 255 103 108 115
 98 102 112 118 116
 81  91 113 123 119
 85  91 111 120 121


Vizinhança 3×3 original (ordenada):   [95, 97, 98, 102, 103, 103, 105, 107, 112]
Mediana original:                       103

Vizinhança 3×3 corrompida (ordenada): [95, 97, 98, 102, 103, 103, 107, 112, 255]
Mediana corrompida:                     103  ← ignora o 255
Média  corrompida:                    119.1  ← distorcida pelo 255
```

O exemplo confirma: mesmo com o pixel corrompido a 255, a mediana retorna o valor central correto — o outlier ocupa a última posição na ordenação e é descartado naturalmente.

Por ser baseada em ordenação e não em soma, a mediana possui três propriedades fundamentais que a diferenciam dos filtros lineares:

- **Robusta** ao ruído impulsivo — outliers vão para as extremidades da sequência ordenada e não afetam o valor central;
- **Preservadora de bordas** — transições abruptas de intensidade são mantidas, pois a mediana seleciona um valor que já existe na vizinhança, sem criar novos níveis intermediários;
- **Não linear** — não pode ser expressa como convolução, portanto `mm.conv` não se aplica; usa-se `cv2.medianBlur`.

A Figure 3.20 compara média e mediana aplicadas à imagem do mandrill com 10% de ruído sal e pimenta:

```
# 10% de ruído para evidenciar diferenças entre filtros
noisy_5, _ = add_salt_pepper(img_gray_crop, prob=0.1)

#   Filtros
f_gauss   = cv2.GaussianBlur(noisy_5, (5, 5), 1.0)
f_media   = mm.conv(noisy_5, np.ones((5, 5), dtype=np.float32) / 20.0)
f_median3 = cv2.medianBlur(noisy_5, 3)
f_median5 = cv2.medianBlur(noisy_5, 5)
f_bilat   = cv2.bilateralFilter(noisy_5, d=9, sigmaColor=75, sigmaSpace=75)

# filtros morfológicos no próximo capítulo, mas já adiantados aqui para comparação
# B         = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
# f_morf    = cv2.morphologyEx(           # apresentado no próximo capítulo
#                   cv2.morphologyEx(noisy_5, cv2.MORPH_OPEN,  B),
#                   cv2.MORPH_CLOSE, B)
B = mm.secross()  # elemento estruturante de caixa 3×3
f_morf     = mm.asf(noisy_5, 'OC', B)  # equivalente via morph

#   MSE
def mse(a, b):
    return float(np.mean((a.astype(np.float32) - b.astype(np.float32))**2))

print(f"{'Filtro':<22} {'MSE':>8}")
print("-" * 32)
pares = [("Gaussiano 5×5",    f_gauss),
         ("Média 5×5",        f_media),
```

```
        ("Mediana 3×3",      f_median3),
        ("Mediana 5×5",      f_median5),
        ("Bilateral",        f_bilat),
        ("Morf. open+close", f_morf)]
for nome, img in pares:
    print(f"{nome:<22} {mse(img_gray_crop, img):>8.2f}")

imgs   = [img_gray_crop, noisy_5,    f_gauss,        f_media,
          f_median3,     f_median5,  f_bilat,        f_morf]
titles = ["Original",    "Ruído 10%", "Gaussiano 5×5","Média 5×5",
          "Mediana 3×3", "Mediana 5×5", "Bilateral", "Morf. open+close"]

mm.show(
    imgs,
    titles=[f"Crop: {t}" for t in titles],
    cols=4, rows=2, figsize=(14, 8)
)
```

```
Filtro                          MSE
--------------------------------
Gaussiano 5×5                264.73
Média 5×5                    876.54
Mediana 3×3                   33.43
Mediana 5×5                   70.11
Bilateral                   1012.00
Morf. open+close              67.84
```

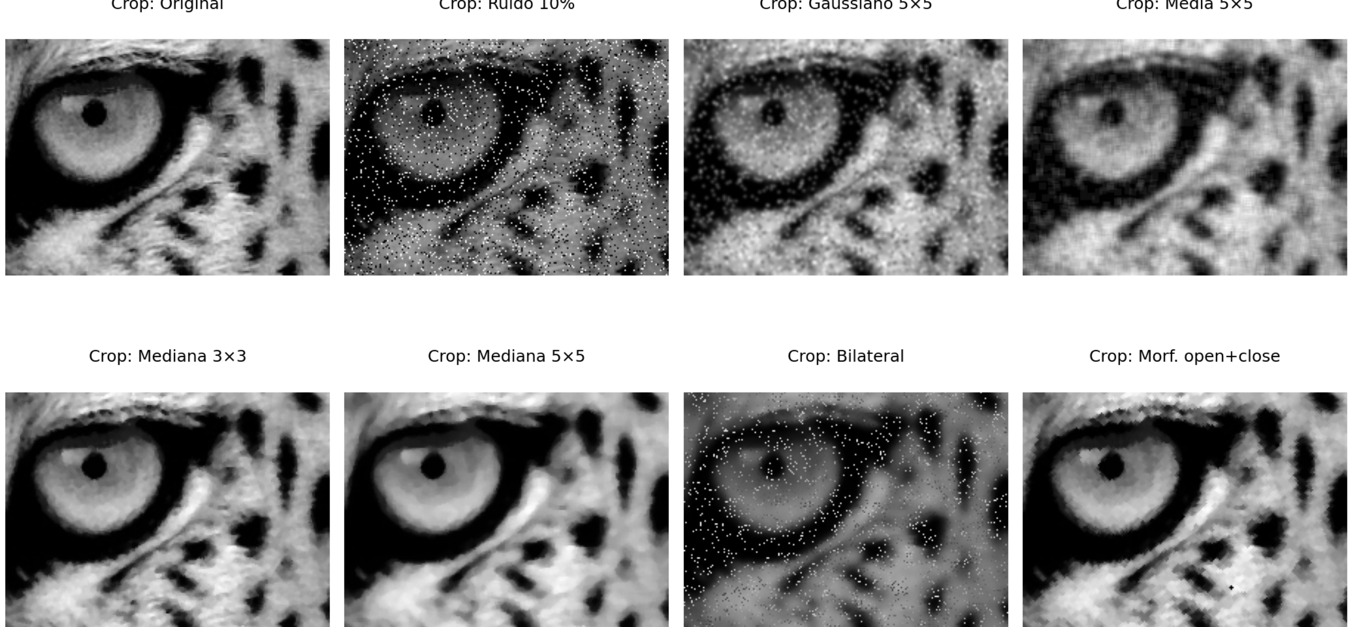


Figura 3.20: Comparação de filtros para remoção de ruído sal e pimenta (10%): Gaussiano, Média, Mediana, Bilateral e Morfológico. Linha superior: imagens completas; linha inferior: crop da região de interesse.

3.8 Aplicação Prática: Pré-processamento para Segmentação

Na prática, as técnicas deste capítulo raramente são usadas isoladamente. Um **pipeline de pré-processamento** típico combina várias etapas em sequência, adaptando-se ao tipo de imagem e à aplicação. A Figure 3.21 ilustra um pipeline completo:

1. **Equalização de histograma (CLAHE):** normaliza o contraste independente das condições de iluminação;

2. **Filtro Gaussiano:** suaviza ruído de aquisição sem destruir bordas;
3. **Detecção de bordas (Sobel/Canny):** extrai estruturas relevantes para segmentação.

> **Ordem importa**
>
> A ordem das operações afeta o resultado final. Em geral: **(1) normalização de intensidade → (2) redução de ruído → (3) realce/segmentação**. Inverter a ordem pode amplificar ruído ou perder bordas antes de detectá-las.

```
# Etapa 1: CLAHE
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
img_clahe = clahe.apply(img_gray_crop)

# Etapa 2: Gaussiano
img_gauss = cv2.GaussianBlur(img_clahe, (5, 5), 0)

# Etapa 3: Canny
edges = cv2.Canny(img_gauss, 50, 150)

# Comparação: pipeline vs. Canny direto
edges_direct = cv2.Canny(img_gray_crop, 50, 150)

mm.show(
    [img_gray_crop, img_clahe, img_gauss, edges, edges_direct],
    titles=["Original", "1. CLAHE", "2. Gaussiano", "3. Canny (pipeline)", "Canny (direto)"],
    cols=5
)
```

Figura 3.21: Pipeline de pré-processamento: CLAHE → Gaussiano → Canny. Cada etapa prepara a imagem para a seguinte, resultando em bordas limpas e bem definidas.

3.9 Resumo

Neste capítulo foram apresentadas as principais técnicas de processamento no domínio espacial, da manipulação direta de pixels até a filtragem por vizinhança:

- **Operações de ponto:** aritméticas saturadas (`mm.addm`, `mm.subm`) e lógicas bit a bit (`mm.band`, `mm.bor`, `mm.bnot`) para recorte de ROI e combinação de imagens; *alpha blending* (`mm.blend`) para fusão ponderada com peso $\alpha \in [0, 1]$.
- **Histograma:** função discreta de distribuição de intensidades; visualizado com `mm.histImg` e calculado com `mm.hist`; base para diagnóstico tonal e para as técnicas de equalização e especificação.
- **Equalização:** redistribuição automática das intensidades pela CDF (`mm.equalize`), com variante adaptativa CLAHE para controle local do contraste.
- **Especificação de histograma:** transferência do perfil tonal de uma imagem de referência via mapeamento inverso da CDF — generalização da equalização para distribuições arbitrárias.
- **Correlação e convolução:** mecanismo de janela deslizante implementado em `mm.conv` (`cv2.filter2D`); diferenciados pela rotação de 180° do kernel — relevante apenas para kernels assimétricos.

- **Filtros de suavização:** média (kernel uniforme, borra bordas proporcionalmente ao tamanho) e Gaussiano (ponderação radial, separável, sem *ringing*, preserva melhor as bordas).
- **Filtros de realce:** Laplaciano (w_4/w_8, segunda derivada isotrópica), Sobel (gradiente direcional de primeira ordem, com magnitude $|\nabla f|$ e direção θ) e *Unsharp Masking* (amplificação das altas frequências com parâmetro k).
- **Filtro da mediana:** não linear, robusto a outliers, preserva bordas — superior aos filtros lineares para ruído sal e pimenta.
- **Pipeline prático:** encadeamento CLAHE → Gaussiano → Canny como estratégia de pré-processamento; `mm.drawImageKernel` para visualização didática da janela deslizante.

O Capítulo 4 abordará o **processamento no domínio da frequência** (Transformada de Fourier) e a **morfologia matemática** (erosão, dilatação, abertura, fechamento) — onde as funções `mm.ero` e `mm.dil` da biblioteca `morph.py` serão exploradas em profundidade.

3.10 🤖 Uso do NotebookLM como Tutor Complementar

Nesta edição, incentivamos o uso do **NotebookLM** como ferramenta complementar de aprendizagem. Essa ferramenta de IA utiliza exclusivamente os documentos fornecidos pelo autor como base de conhecimento, garantindo respostas coerentes com o conteúdo do livro — incluindo as funções da biblioteca `morph.py` e os experimentos realizados neste capítulo.

Para cada capítulo, preparamos um projeto específico na plataforma com o PDF do capítulo, os notebooks e materiais auxiliares. Sugerimos explorar especialmente:

- **Guia de Estudo:** resumo estruturado dos conceitos, ideal para revisão antes de provas;
- **Conversa:** tire dúvidas sobre equalização, convolução, filtros e pipelines diretamente com o tutor;
- **Perguntas frequentes:** questões típicas sobre a diferença entre média e mediana, USM, Laplaciano vs. Sobel.

🎓 Estude com o Tutor Inteligente

Para interagir com o conteúdo deste capítulo, acesse o link a seguir. O ambiente contém materiais didáticos em diferentes formatos, gerados a partir do **PDF** do capítulo. Na plataforma, explore especialmente as opções **Guia de Estudo** e **Conversa** para aprofundar sua compreensão.
🚀 ACESSAR NOTEBOOKLM: CAPÍTULO 03

3.11 Lista de Exercícios

1. **(10%)** Explique a diferença entre **convolução** e **correlação cruzada**. Para quais tipos de *kernel* os resultados são idênticos? Dê um exemplo de *kernel* assimétrico (como Sobel G_x) e mostre numericamente que os resultados diferem aplicando-o ao patch 5×5 do capítulo das duas formas.

2. **(15%)** Considere uma imagem 5×5 com intensidades concentradas entre os níveis 3 e 5 (baixo contraste, 3 bits). Aplique manualmente o algoritmo de equalização da Table 3.1, preenchendo todas as colunas da tabela (k, $h[k]$, $p[k]$, $\text{cdf}[k]$, $\text{lut}[k]$). Verifique o resultado com `mm.equalize`.

3. **(15%)** Usando `mm.conv`, aplique o filtro de média com kernels de tamanho 3×3, 9×9 e 21×21 à imagem do mandrill. Para cada versão, calcule o **PSNR** (*Peak Signal-to-Noise Ratio*) em relação à original:

$$\text{PSNR} = 10\log_{10}\left(\frac{255^2}{\text{MSE}}\right), \quad \text{MSE} = \frac{1}{MN}\sum_{i,j}(f-g)^2$$

   Plote o PSNR em função do tamanho do kernel e explique o que a queda progressiva indica sobre a relação entre suavização e perda de informação.

4. **(15%)** Usando `add_salt_pepper` com densidade de 5%, aplique e compare: (a) `mm.conv` com média 3×3, (b) `cv2.GaussianBlur` com $\sigma = 1$, (c) `cv2.medianBlur` com janela 3×3 e (d) `cv2.medianBlur` com janela 5×5. Exiba as imagens com `mm.show` em grade 2×4 (linha 1: imagens, linha 2: histogramas via `mm.histImg`). Explique por que a mediana supera os filtros lineares usando o argumento da Table 3.4.
5. **(15%)** Implemente `mm.conv0` usando apenas operações NumPy vetorizadas — sem laços Python e sem `cv2.filter2D` — com o operador de *stride tricks* (`np.lib.stride_tricks.sliding_window_view`). Compare o resultado e o tempo de execução com `mm.conv0` (laços) e `mm.conv` (cv2) para kernels 3×3 e 15×15 na imagem do mandrill.
6. **(15%)** Aplique o *Unsharp Masking* com $\sigma = 1$ e $k \in \{0.5, 1.0, 2.0, 4.0\}$ usando a função `usm` do capítulo. Para cada valor de k: (a) calcule a diferença absoluta $|g - f|$, (b) exiba as imagens e as diferenças com `mm.show`, e (c) plote o histograma das diferenças com `mm.histImg`. Identifique a partir de qual k os artefatos (halos e amplificação de ruído) tornam-se visualmente inaceitáveis.
7. **(15%)** Escolha uma imagem de raio-X ou tomografia disponível publicamente (ex.: via `mm.read` de URL) e projete um pipeline de pré-processamento com pelo menos 4 etapas sequenciais, justificando cada escolha com base nos conceitos do capítulo. Exiba com `mm.show` em grade: imagem original, cada etapa intermediária e o resultado final com seus histogramas (`mm.histImg`).

Referências do Capítulo

A fundamentação teórica deste capítulo baseia-se nas seguintes obras:

- Gonzalez; Woods (2018) para os conceitos de operações de intensidade, histograma, convolução e filtragem espacial.
- Szeliski (2022) para a visão computacional e aplicações práticas de filtragem.
- Bradski; Kaehler (2008) para a implementação prática com OpenCV e `morph.py`.

3.12 💻 Parte Prática com Exercícios de Programação

🎯 Objetivo deste Caderno

O caderno permite desenvolver, validar, organizar e testar soluções de **Exercícios de Programação (EPs)** em ambientes interativos, como o Colab, com os mesmos casos de teste do Moodle, copiando para lá apenas na hora de registrar a nota oficial.

Download

Baixe `morph.py` e `testsuite.py` executando a célula abaixo:

```
import os, sys, importlib, inspect, urllib.request

# URLs do repositório
BASE_URL = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph"
for f in ["morph.py", "testsuite.py"]:
    if not os.path.exists(f):
        urllib.request.urlretrieve(f"{BASE_URL}/{f}", f)

import morph, testsuite
```

```
importlib.reload(morph); importlib.reload(testsuite)
from morph import mm
from testsuite import TestSuite

print(f" Ambiente pronto. Morph: {morph.__version__} | TestSuite: {testsuite.__version__}")
```

```
 Ambiente pronto. Morph: 1.1.0 | TestSuite: 1.1.0
```

Executando os Testes

Para rodar os testes, execute `TestSuite("EP01_01.extensão").run()` numa nova célula, trocando a extensão pela da linguagem usada (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). O sistema baixa os casos de teste do GitHub, executa o programa e calcula a nota automaticamente.

Exemplo de teste de `sqrt` em Python, com `timeit` isolando cada operação:

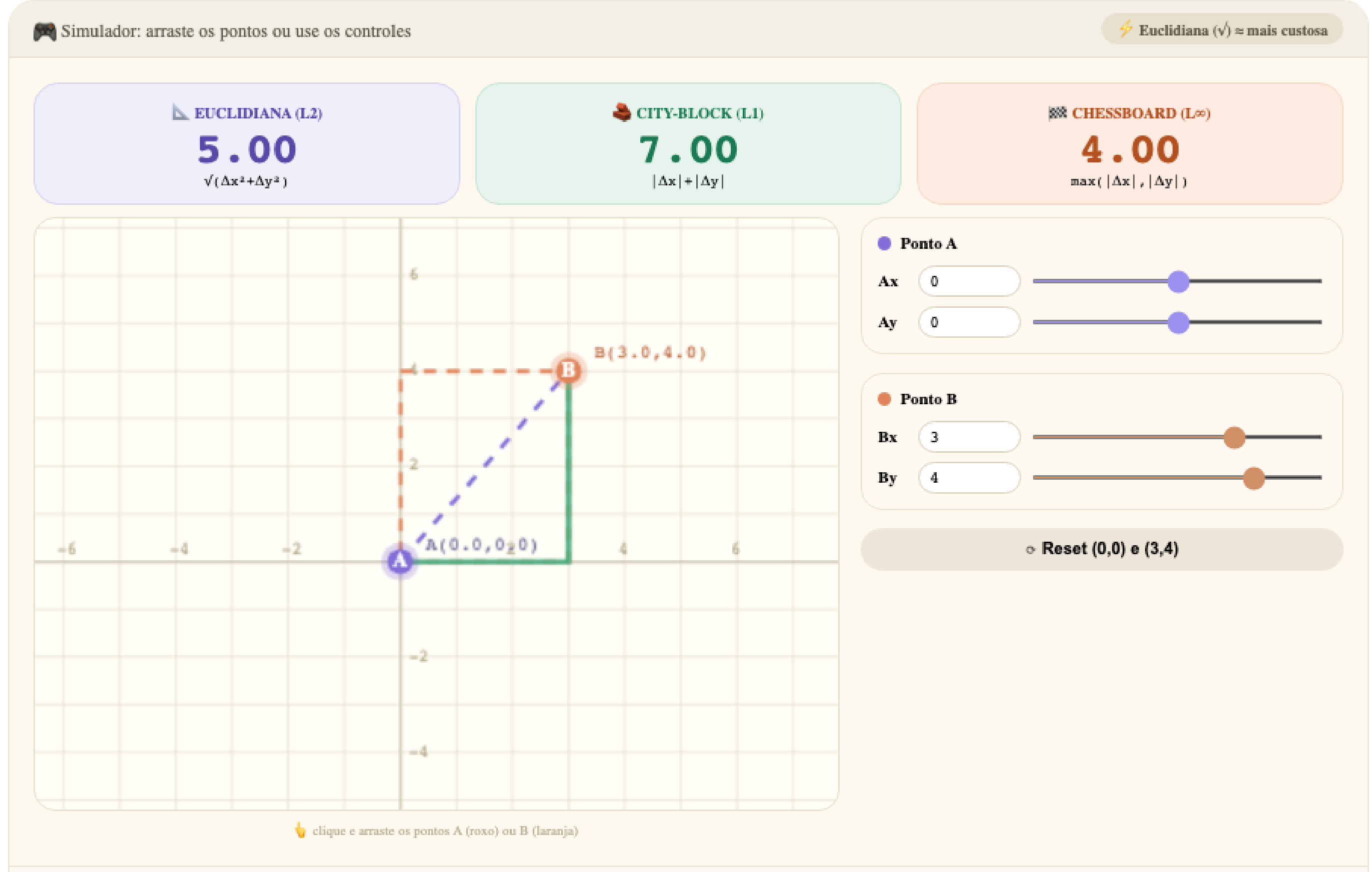


Figura 3.22: Simulador: Distâncias Euclidiana, *City-block* e *Chessboard*

```
%%writefile EP02_01.py
# Código Python
x1,y1,x2,y2 = int(input()), int(input()), int(input()), int(input())
# Cálculo das diferenças
dx = abs(x2 - x1)
dy = abs(y2 - y1)

# 1. Distância Euclidiana (L2)
dist_euclidiana = (dx**2 + dy**2)**0.5

# 2. Distância City-block / Manhattan (L1)
```

```
dist_city_block = dx + dy

# 3. Distância Chessboard / Chebyshev (Linf)
dist_chessboard = max(dx, dy)

# Saída formatada conforme os casos de teste
print(f"{dist_euclidiana:.2f}")
print(f"{dist_city_block:.2f}")
print(f"{dist_chessboard:.2f}")
```

```
Writing EP02_01.py
```

```
import numpy as np
np.set_printoptions(linewidth=100, edgeitems=3, threshold=1000)
```

```
print("Substitui emoji Unicode literal  por comando")
```

```
Substitui emoji Unicode literal  por comando
```

```
import numpy as np
print(mm.drawImage(np.zeros((5, 5))))
```

```
  0   0   0   0   0
  0   0   0   0   0
  0   0   0   0   0
  0   0   0   0   0
  0   0   0   0   0
```

```
TestSuite("EP02_01.py").run()
```

```
from IPython.display import display, HTML
display(HTML("<h1>oi</h1>"))
```

Capítulo 4

Morfologia Matemática e Segmentação de Imagens

Este capítulo apresenta dois temas fundamentais do Processamento Digital de Imagens (PDI): a **morfologia matemática** e a **segmentação de imagens**. A morfologia matemática fornece um arcabouço teórico baseado na teoria dos conjuntos para analisar, refinar e quantificar a forma de objetos em imagens binárias e em tons de cinza, a partir de operadores fundamentais como **erosão** e **dilatação**. A segmentação, por outro lado, tem como objetivo particionar a imagem em regiões semanticamente relevantes, separando objetos do fundo e identificando estruturas de interesse.

O capítulo inicia com a **limiarização**, uma das técnicas mais simples e importantes de segmentação, introduzindo o método automático de **Otsu** e a análise do histograma por meio da variância interclasses, apresentada inicialmente no primeiro capítulo. Em seguida, são apresentados os principais operadores da morfologia matemática, com destaque para *pipelines* de limpeza binária que combinam preenchimento de regiões, erosão e dilatação. Por fim, técnicas mais avançadas de segmentação, como o algoritmo da **transformada de distância**, ***watershed*** e o agrupamento por ***k-means***, ampliam a caixa de ferramentas para separação e análise de objetos em imagens.

4.1 Objetivos

Ao final deste capítulo, você será capaz de:

- **Aplicar limiarização:** Compreender o critério automático de Otsu por maximização da variância interclasses (σ_B^2) e escolher o melhor pré-processamento por análise de bimodalidade do histograma;
- **Dominar morfologia binária:** Compreender e aplicar erosão ($A \ominus B$) e dilatação ($A \oplus B$) como primitivos espaciais, derivando a abertura ($A \circ B$), o fechamento ($A \bullet B$) e as operações baseadas em reconstrução morfológica (`mm.clohole` e `mm.edgeoff`);
- **Aplicar morfologia em tons de cinza:** Extrair bordas e detalhes estruturais utilizando o gradiente morfológico, *top-hat* e *black-hat*;
- **Segmentar por regiões:** Construir o *pipeline* de segmentação por ***watershed*** topográfico baseado em marcadores extraídos via **transformada de distância**;
- **Segmentar por agrupamento:** Utilizar o algoritmo ***k-means*** associado ao método do cotovelo para otimização do número de classes k;
- **Extrair descritores de forma:** Rotular componentes conexas e quantificar propriedades geométricas (área, perímetro, circularidade, excentricidade e momentos de Hu) via `cv2.findContours`.

```
import os, importlib, urllib.request
import numpy as np
import matplotlib.pyplot as plt
import cv2
from scipy import ndimage
```

```
BASE_URL = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph"
for f in ["morph.py"]:
    if not os.path.exists(f):
        urllib.request.urlretrieve(f"{BASE_URL}/{f}", f)

import morph
importlib.reload(morph)
from morph import mm

print("  Ambiente pronto")
```

```
 Ambiente pronto
```

4.2 Limiarização

A **limiarização** (*thresholding*) é uma das formas mais simples e eficientes de segmentação de imagens. Seu objetivo é classificar cada pixel em duas classes de intensidade, normalmente associadas a *objeto* e *fundo*:

$$g(x,y) = \begin{cases} 255, & \text{se } f(x,y) \geq T \\ 0, & \text{caso contrário} \end{cases} \tag{4.1}$$

em que $f(x,y)$ representa a intensidade do pixel na imagem original e $g(x,y)$ a imagem binária resultante.

A escolha do limiar T é importante para a qualidade da segmentação. O método de **Otsu** determina automaticamente o limiar ótimo ao maximizar a **variância interclasses** σ_B^2 e definida por:

$$\sigma_B^2(T) = w_0(T)\, w_1(T)\, \left[\mu_0(T) - \mu_1(T)\right]^2 \tag{4.2}$$

em que:

- $w_0(T)$ e $w_1(T)$ são as probabilidades acumuladas das classes fundo e objeto;
- $\mu_0(T)$ e $\mu_1(T)$ são as médias de intensidade dessas classes;
- $\sigma_B^2(T)$ representa a variância interclasses para um dado limiar T.

O método funciona melhor quando o histograma apresenta dois grupos de intensidades relativamente separados. Para isso, o algoritmo avalia todos os limiares possíveis da imagem — tipicamente no intervalo $[0, 255]$ para imagens de 8 bits — e seleciona o valor que maximiza a variância entre classes, denotada por σ_B^2:

$$T^* = \arg \max_{T \in [0,255]} \sigma_B^2(T)$$

Otsu assume histogramas bimodais

O método de Otsu produz melhores resultados quando o histograma apresenta dois picos bem definidos (*bimodalidade*), correspondentes ao fundo e ao objeto. Quanto maior a separação entre esses picos e mais pronunciado o máximo de σ_B^2, mais confiável tende a ser o limiar obtido.
Em imagens com iluminação não uniforme ou múltiplas regiões de intensidade, técnicas de **limiarização adaptativa** — nas quais o limiar é calculado localmente — costumam produzir segmentações mais robustas.
O índice B em σ_B^2 significa ***between classes*** (*entre classes*). Assim, σ_B^2 representa a **variância entre as classes** (*between-class variance*).

4.2.1 Imagem de Moedas

A imagem utilizada para praticar a segmentação é uma fotografia de coleção de moedas de diferentes países e épocas (Figure 4.1). Crédito: GAZI.MD.AHAD (CC BY-SA 4.0). Ela apresenta objetos circulares com bordas bem definidas, sendo ideal para demonstrar limiarização, operadores morfológicos, transformada de distância, *watershed* e descritores de forma.

```
import os

url     = "https://upload.wikimedia.org/wikipedia/commons/2/25/GAZI.MD.AHAD_11.jpg"
caminho = "imagens/coins.jpg"

if not os.path.exists(caminho):
    os.makedirs("imagens", exist_ok=True)
    img_obj = mm.read(url, pil=True)
    mm.write(img_obj, caminho)
else:
    img_obj = mm.read(caminho, pil=True)

img_coins_color = np.array(img_obj)
img_coins_gray  = mm.gray(img_coins_color)

print(f"Dimensões [y,x,c]: {img_coins_color.shape}")
mm.show(img_coins_color, scale=30)
```

```
Dimensões [y,x,c]: (2560, 1920, 3)
```

Figura 4.1: Imagem com moedas de vários tipos. Crédito: GAZI.MD.AHAD (CC BY-SA 4.0).

4.2.2 Escolha do Pré-processamento para o Otsu

A qualidade do Otsu depende diretamente de quão **bimodal** é o histograma da imagem de entrada. A imagem original de moedas tem iluminação não uniforme e moedas escuras (cobre oxidado) próximas em intensidade ao fundo escuro, tornando o histograma pouco bimodal.

Para corrigir essas limitações, serão avaliadas duas técnicas clássicas de pré-processamento, apresentadas no capítulo anterior, aplicadas antes da etapa de limiarização:

Técnica	O que faz	Quando usar
CLAHE	Equalização de histograma adaptativa local	Baixo contraste global ou regional
Gaussiano	Suavização por convolução com gaussiana	Ruído de alta frequência (textura do fundo)

A função `cv2.createCLAHE(clipLimit, tileGridSize)` divide a imagem em blocos e aplica equalização de histograma em cada um, limitando a amplificação de ruído pelo parâmetro `clipLimit`. O filtro Gaussiano (`cv2.GaussianBlur`) suaviza texturas finas que poderiam criar falsos picos no histograma.

Para comparar objetivamente qual pré-processamento gera melhor entrada para o Otsu, plota-se para cada versão: a imagem, o histograma com T^* marcado, a curva $\sigma_B^2(T)$ e o resultado da binarização.

Critério de comparação

A versão com **maior valor de σ_B^2 no pico** é a que oferece melhor separação bimodal — e portanto a melhor entrada para o Otsu.

```python
import cv2, io

def otsu_criterio(img):
    h=mm.hist(img); p=h/h.sum();                     # histograma e probabilidades
    sigma2=np.zeros(len(p))
    for T in range(1,len(p)):                        # percorre limiares
        w0,w1=p[:T].sum(),p[T:].sum()                # probabilidades das classes
        if w0*w1==0: continue                        # evita divisão por zero
        mu0=(np.arange(T)*p[:T]).sum()/w0            # média fundo
        mu1=(np.arange(T,len(p))*p[T:]).sum()/w1     # média objeto
        sigma2[T]=w0*w1*(mu0-mu1)**2                 #  ²B(T)
    return sigma2,np.argmax(sigma2)                  # curva e T ótimo

def fig2img(fig):
    b=io.BytesIO(); fig.savefig(b,format='png',dpi=100)   # figura → buffer
    plt.close(fig); b.seek(0)
    return np.array(plt.imread(b))                         # buffer → array

def plot_curve(y,T,title,ylabel,color):
    fig,ax=plt.subplots(figsize=(4,3))
    ax.plot(y,color=color) if ylabel==" ²B" else ax.bar(range(len(y)),y,color=color,width=1)
    ax.axvline(T,color='red',lw=2,label=f"T*={T}")        # limiar ótimo
    ax.set(title=title,xlabel="T" if ylabel==" ²B" else "Intensidade",ylabel=ylabel)
    ax.legend(fontsize=8); plt.tight_layout()
    return fig2img(fig)

#   Pré-processamentos
clahe=cv2.createCLAHE(clipLimit=2.0,tileGridSize=(8,8))
img_clahe = clahe.apply(img_coins_gray)
img_gauss = cv2.GaussianBlur(clahe.apply(img_coins_gray),(5,5),0)
imgs0=[("Original",img_coins_gray),
       ("CLAHE",img_clahe),
       ("CLAHE+Gauss",img_gauss)]

#   Tabela comparativa
print(f"{'Versão':<18}{'T*':>6}{' ²B pico':>14}")
```

```
print("-"*40)

imgs,titles=[],[]
for nome,img in imgs0:
    sigma2,T=otsu_criterio(img)                               # calcula  ²B(T)
    print(f"{nome:<18}{T:>6}{sigma2[T]:>14.4e}")
    imgs += [
        img,                                                  # imagem
        plot_curve(mm.hist(img),T,f"Hist T*={T}","Freq.","steelblue"),
        plot_curve(sigma2,T," ²B(T)"," ²B","darkorange"),
        mm.threshold(img)                                     # Otsu final
    ]
    titles += [nome,f"Hist T*={T}"," ²B(T)",f"Otsu T*={T}"]

#   Exibição final
mm.show(imgs,titles=titles,cols=4,figsize=(12,12),dpi=200)
```

```
Versão                T*       ²B pico
----------------------------------------
Original             105    1.9437e+03
CLAHE                122    2.4695e+03
CLAHE+Gauss          123    2.4283e+03
```

Figura 4.2: Comparação dos pré-processamentos: imagem | histograma+T* | ^{2}B(T) | Otsu. A melhor separação bimodal indica o limiar mais confiável.

4.2.3 Resultado: CLAHE como Melhor Pré-processamento

A análise da Figure 4.2 mostra que o **CLAHE** produz a maior separação bimodal ($\sigma_B^2 \approx 2{,}47 \times 10^3$, $T^* = 122$), superando até mesmo a combinação CLAHE+Gaussiano. O filtro Gaussiano, embora suavize ruídos de textura, reduz levemente a separação inter-classe ao homogeneizar intensidades próximas ao limiar.

Interpretação visual

No histograma do CLAHE, observe o **vale profundo** em torno de $T^* = 122$, separando claramente dois grupos: pixels de fundo (intensidades baixas) e pixels de moedas (intensidades altas). Na curva $\sigma_B^2(T)$, o pico alto e estreito confirma que esse limiar é robusto.

4.3 Morfologia Matemática

A **morfologia matemática** é uma teoria baseada em conjuntos utilizada para analisar a forma e a estrutura de objetos em imagens. Diferentemente dos filtros lineares apresentados no Capítulo 3, os operadores morfológicos são **não lineares**, pois se baseiam em operações de mínimo, máximo e inclusão espacial, em vez de combinações lineares de intensidades. Esses operadores atuam sobre a vizinhança de cada pixel por meio de um **elemento estruturante** $\mathbb{B}$, responsável por definir a forma e o tamanho da região analisada.

Em imagens binárias, o elemento estruturante transladado para a posição x é definido como:

$$\mathbb{B}_x = \{x + b \mid b \in \mathbb{B}\} \cap \mathbb{E}, \quad x \in \mathbb{E}$$

A interseção com $\mathbb{E}$ garante que apenas as posições dentro do domínio da imagem sejam consideradas — quando x está próximo à borda, parte de $\mathbb{B}_x$ pode cair fora de $\mathbb{E}$ e é simplesmente descartada.

Quando o elemento estruturante associa pesos a seus elementos — isto é, $b : \mathbb{B} \to \mathbb{Z}$ — ele é denominado **função estruturante** ou **elemento estruturante não plano**.

Desenvolvida por Matheron e Serra na década de 1960 para imagens binárias e posteriormente estendida a tons de cinza, a morfologia matemática fundamenta operadores como gradiente morfológico, *top-hat*, *watershed* e transformada de distância, todos derivados de dois primitivos: **erosão** e **dilatação** (Matheron, 1975; Serra, 1982).

4.3.1 Erosão e Dilatação

Os dois operadores primitivos são definidos de forma unificada para imagens em tons de cinza ($f : \mathbb{E} \to \mathbb{Z}$) e, por restrição ao domínio $\{0, 1\}$, também para imagens binárias.

4.3.1.1 Erosão

A **Erosão** de f pela função estruturante $b : \mathbb{B} \to \mathbb{Z}$ é definida por:

$$\varepsilon_b(f)(x) = (f \ominus b)(x) = \min_{y \in \mathbb{B}_x} \{\, f(y) - b(y - x) \,\}, \quad \forall\, x \in \mathbb{E} \tag{4.3}$$

Na prática, a erosão substitui cada pixel pelo menor valor encontrado em sua vizinhança.

O valor no pixel x é o **mínimo** de $f(y) - b(y - x)$ sobre a vizinhança $\mathbb{B}_x$. Valores positivos no elemento estruturante forçam o resultado para baixo, 'cavando' a imagem mais profundamente e favorecendo a erosão. No caso **plano** ($b \equiv 0$), a expressão reduz-se a:

$$\varepsilon_B(f)(x) = \min\{\, f(y) : y \in \mathbb{B}_x \,\}$$

e em imagens binárias equivale a exigir que $\mathbb{B}$, transladado para x, esteja **completamente contido** em A:

$$A \ominus \mathbb{B} = \{\, z \in \mathbb{E} \mid \mathbb{B}_z \subseteq A \,\}$$

Efeito: *encolhe* objetos, suprimindo protuberâncias menores que $\mathbb{B}$.

4.3.1.2 Dilatação

A **Dilatação** de f pela função estruturante $b : \mathbb{B} \to \mathbb{Z}$ é definida por:

$$\delta_b(f)(x) = (f \oplus b)(x) = \max_{y \in \mathbb{B}_x} \{\, f(y) + b(x - y) \,\}, \quad \forall\, x \in \mathbb{E} \tag{4.4}$$

Na prática, a dilatação substitui cada pixel pelo maior valor presente na vizinhança definida pelo elemento estruturante.

O valor no pixel x é o **máximo** de $f(y) + b(x - y)$ sobre a vizinhança $\mathbb{B}_x$: o argumento $x - y$ corresponde a avaliar o transposto $\hat{b}$, o que é consistente com a dualidade erosão–dilatação. No caso **plano** ($b \equiv 0$), a expressão reduz-se a:

$$\delta_B(f)(x) = \max\{\, f(y) : y \in \mathbb{B}_x \,\}$$

e em imagens binárias equivale a exigir que $\hat{\mathbb{B}}$, transladado para x, **intersecte** A:

$$A \oplus B = \{\, z \in \mathbb{E} \mid \hat{\mathbb{B}}_z \cap A \neq \emptyset \,\}$$

Efeito: *expande* objetos, preenchendo lacunas menores que B.

Dualidade erosão–dilatação

Erosão e dilatação são **duais pelo complemento**. Isso significa que um operador pode ser completamente obtido a partir do outro, desde que se atue sobre o complemento da imagem utilizando o elemento estruturante refletido $\hat{B}$:

$$(A \ominus B)^c = A^c \oplus \hat{B} \iff A \ominus B = (A^c \oplus \hat{B})^c$$

De forma análoga, a **dilatação também pode ser obtida a partir da erosão**:

$$(A \oplus B)^c = A^c \ominus \hat{B} \iff A \oplus B = (A^c \ominus \hat{B})^c$$

Em termos práticos, erodir o objeto isolado equivale a dilatar o seu plano de fundo (e vice-versa). Na implementação do pacote `morph.py`, as versões didáticas `mm.ero0` e `mm.dil0` tornam explícita essa estrutura por meio de laços (*loops*) explícitos, enquanto `mm.ero` e `mm.dil` delegam as operações ao OpenCV visando eficiência computacional.

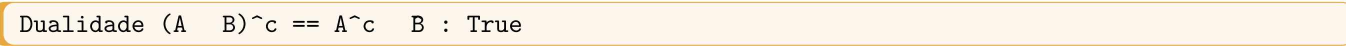
Condições de contorno e restrições finitas

Na teoria matemática pura (em espaços contínuos ou discretos infinitos $\mathbb{Z}^2$), essa equivalência é absoluta. Contudo, no ambiente computacional sobre matrizes finitas, existem duas restrições para que a igualdade seja estritamente válida:

1. **Equivariância por translação:** O operador deve ser equivariante por translação — isto é, deslocar a entrada e depois aplicar o operador deve produzir o mesmo resultado que aplicar o operador e depois deslocar a saída. Quando o elemento estruturante B varia com a posição do pixel (morfologia variante no espaço), essa propriedade é quebrada e a dualidade deixa de valer em geral.
2. **Condições de contorno:** Ao processar as bordas da matriz, o comportamento atribuído aos pixels externos à imagem influi diretamente no resultado. Para que a dualidade se mantenha, a estratégia de preenchimento de borda da erosão deve ser o complemento exato da adotada na dilatação: se a erosão assume que o exterior é preenchido (255), a dilatação correspondente deve assumir que o exterior de A^c é vazio (0).

Para ilustrar numericamente os operadores morfológicos e a dualidade erosão–dilatação, a Figure 4.3 apresenta uma imagem binária 10×10 processada com um elemento estruturante em formato de "L". Na implementação de `morph.py`, a origem de B é fixada no centro geométrico da máscara — posição $(1, 1)$ para um *kernel* 3×3 — e deve ser um elemento ativo para que a erosão se comporte corretamente (conforme discutido anteriormente). O B_L definido a seguir satisfaz essa condição. A Figure 4.4 complementa a análise com um simulador interativo da erosão, permitindo visualizar o deslocamento do elemento estruturante sobre a imagem e identificar as posições em que ele permanece completamente contido no objeto.

```
A = np.array([
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,1,1,1,0,0,0,0],
    [0,0,1,1,1,1,1,0,0,0],
    [0,1,1,1,1,1,1,1,0,0],
    [0,1,1,1,1,1,1,1,0,0],
    [0,1,1,1,1,1,1,0,0,0],
    [0,0,1,1,1,1,1,1,0,0],
    [0,0,0,1,1,1,1,0,0,0],
    [0,0,0,0,1,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0]], dtype=np.uint8) * 255

B_L = np.array([[1,0,0],
                [1,1,0],
                [1,1,0]], dtype=np.uint8)   # origem no centro (1,1), elemento ativo

# Transposto (reflexão em ambos os eixos): B usado na dilatação da dualidade
B_hat = np.flip(B_L)

# Erosão de A por B_L
img_ero0 = mm.ero0(A, B_L)

# Validação da dualidade: (A   B)^c = A^c   B
A_c   = 255 - A           # complemento de A
ero_c = 255 - img_ero0    # complemento da erosão - lado esquerdo
dil_Ac = mm.dil0(A_c, B_hat)              # lado direito

dualidade_ok = np.array_equal(ero_c, dil_Ac)
print(f"Dualidade (A   B)^c == A^c   B : {dualidade_ok}")

mm.show(
    [A, A_c, img_ero0, ero_c, dil_Ac],
    titles=["A", "A ", "A   B", "(A   B) ", "A   B"],
    cols=5,
    figsize=(15, 3)
)
```



```
Dualidade (A   B)^c == A^c   B : True
```

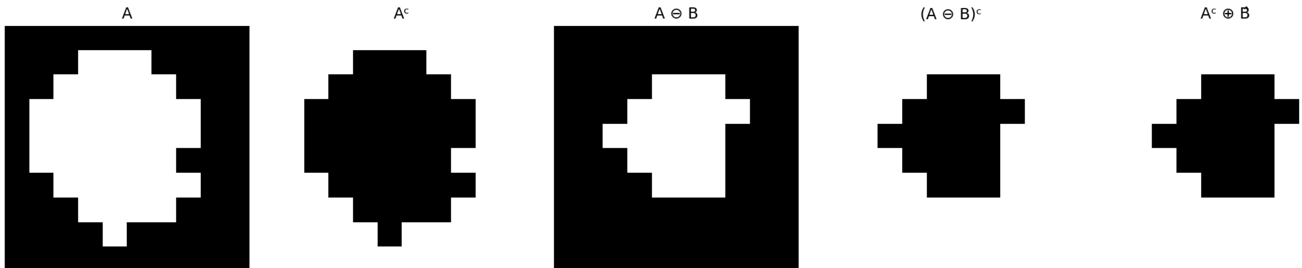


Figura 4.3: Erosão e dilatação em imagem binária 10×10 com elemento estruturante 'L' assimétrico 3×3. Validação da dualidade erosão–dilatação.

4.3.2 Abertura e Fechamento

Combinando erosão e dilatação obtêm-se dois operadores de grande utilidade prática:

Abertura (*opening*) — erosão seguida de dilatação pelo mesmo B:

$$A \circ B = (A \ominus B) \oplus B \tag{4.5}$$

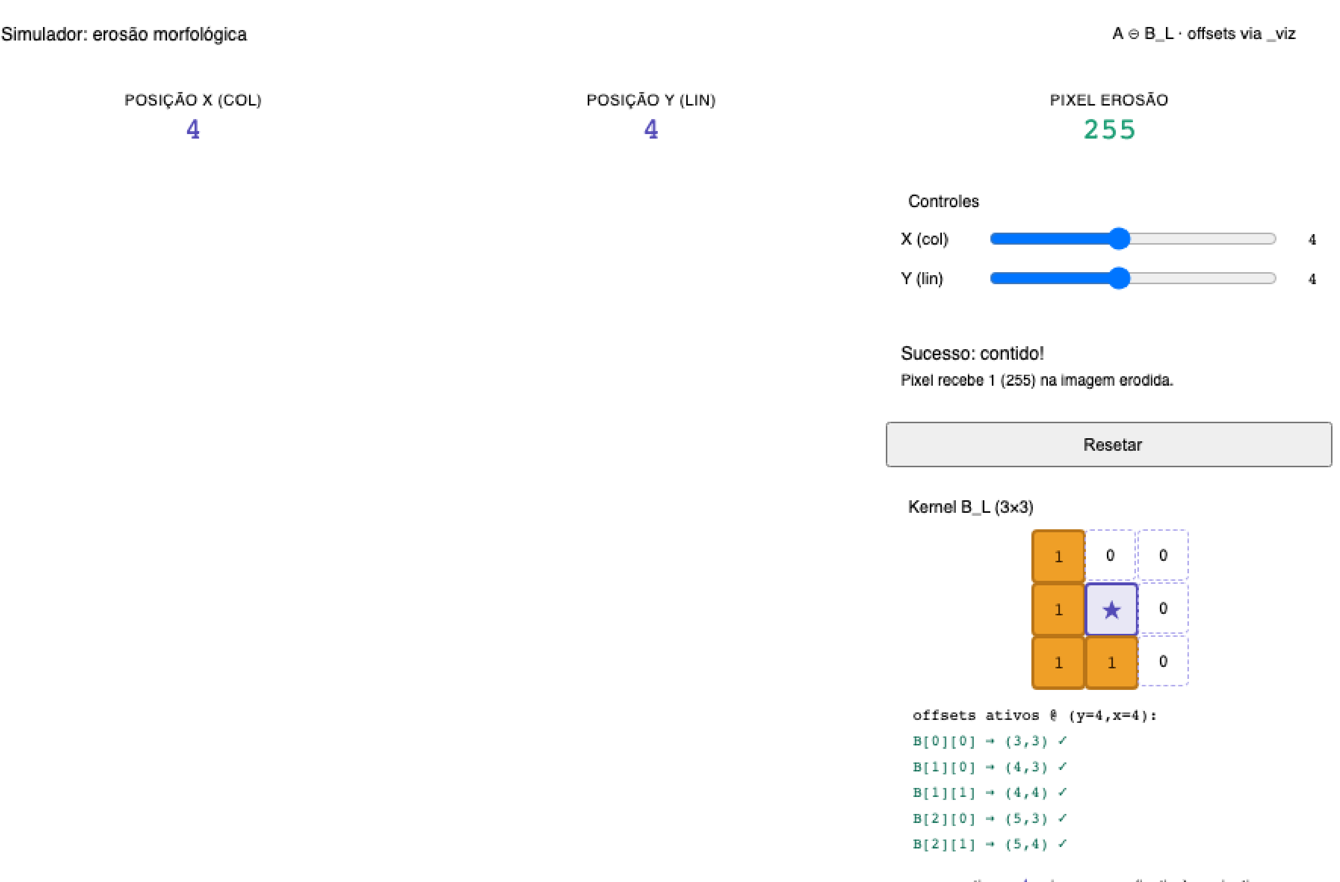


Figura 4.4: Simulador interativo de erosão morfológica: visualização do critério de inclusão do elemento estruturante assimétrico B_L sobre a imagem binária A.

Fechamento (*closing*) — dilatação seguida de erosão pelo mesmo B:

$$A \bullet B = (A \oplus B) \ominus B \tag{4.6}$$

O OpenCV aplica o mesmo `B` nas duas etapas (assume simetria), o que equivale a $(A \ominus B) \oplus B$ para abertura e $(A \oplus B) \ominus B$ para fechamento.

Tabela 4.2: Propriedades de abertura e fechamento.

Operador	Sequência	Efeito principal
Abertura $A \circ B$	erosão → dilatação	Remove estruturas incapazes de conter o elemento estruturante; suaviza contornos externos
Fechamento $A \bullet B$	dilatação → erosão	Preenche buracos menores que B; suaviza contornos internos

Propriedade importante: ambos são **idempotentes** — $(A \circ B) \circ B = A \circ B$ — aplicar o operador duas vezes produz o mesmo resultado que uma vez.

Na prática, abertura e fechamento são frequentemente aplicados em sequência para remover simultaneamente ruído externo e preencher buracos internos. A função `mm.asf` (*Alternating Sequential Filter*) generaliza essa ideia: aplica abertura e fechamento alternadamente, com elemento estruturante crescente a cada iteração i via `mm.sesum(b, i)`. As sequências disponíveis são:

Tabela 4.3: Sequências do filtro sequencial alternado `mm.asf`.

Sequência	Ordem	Uso típico
`'OC'`	abertura → fechamento	remove ruído externo antes de fechar buracos
`'CO'`	fechamento → abertura	fecha buracos antes de remover ruído externo
`'OCO'`	abertura → fechamento → abertura	prioriza remoção de ruído
`'COC'`	fechamento → abertura → fechamento	prioriza preenchimento de buracos

O parâmetro `n` controla o número de iterações — a cada iteração i, o elemento estruturante cresce por soma de Minkowski (`mm.sesum`), o que equivale a aplicar filtros progressivamente maiores. O resultado é uma suavização morfológica multiescala que preserva melhor as estruturas relevantes do que uma única abertura ou fechamento com kernel grande. Ver na Figure 4.5 um exemplo de uso da abertura e fechamento na imagem das moedas.

```
img_bin    = mm.threshold(img_clahe)
B_disk     = mm.sedisk(19)

img_open  = mm.open(img_bin, B_disk)             # erosão → dilatação
img_close = mm.close(img_bin, B_disk)            # dilatação → erosão
img_oc    = mm.close(img_open, B_disk)           # abertura seguida de fechamento
img_asf   = mm.asf(img_bin, 'OC', mm.sedisk(3), n=7)
# ASF: disco base 3×3, cresce a cada iteração

mm.show(
    [img_bin, img_open, img_close, img_oc, img_asf],
    titles=["Binarização Otsu (CLAHE)", "Abertura (A B)", "Fechamento (A B)",
```

```
            "Abertura→Fechamento", "ASF-OC (n=7)"],
    cols=5, figsize=(15, 12)
)
```

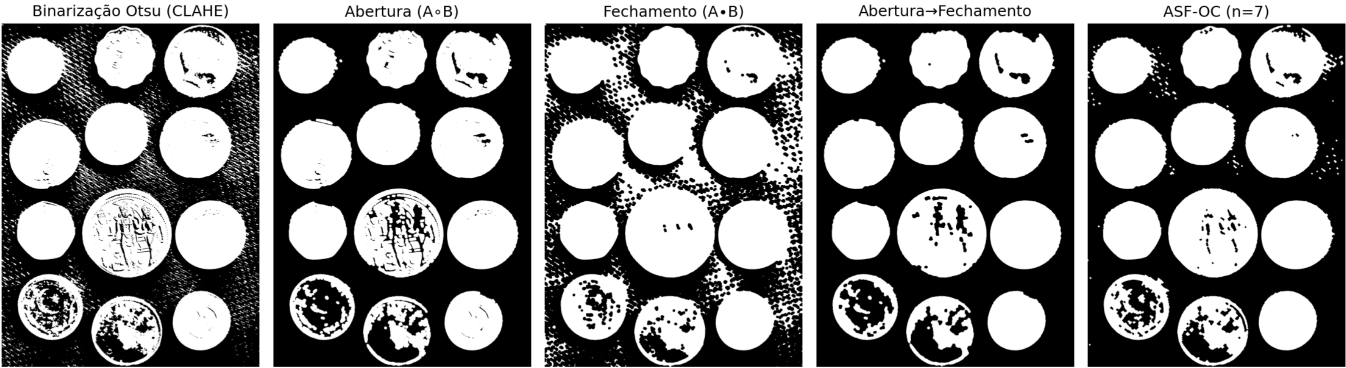


Figura 4.5: Abertura, fechamento, composição e filtro sequencial alternado aplicados à binarização Otsu das moedas com CLAHE. Elemento estruturante: disco 13×13.

4.3.3 Reconstrução Morfológica

A **reconstrução morfológica** propaga uma imagem **marcador** f dentro de uma imagem **máscara** g, garantindo que o resultado nunca ultrapasse g. Define-se a **dilatação geodésica** (condicionada à máscara) como:

$$f \oplus_g b = (f \oplus b) \wedge g$$

onde $\wedge$ denota o mínimo ponto-a-ponto. A operação é iterada até convergência:

$$(f \oplus_g b)^\infty = (\cdots((f \wedge g) \oplus_g b) \oplus_g b \cdots) \oplus_g b$$

produzindo a **reconstrução morfológica** de f sob g:

$$R_g^\delta(f) = (f \oplus_g b)^\infty, \quad \text{onde } (f \oplus_g b)^{(k)} = (f \oplus_g b)^{(k-1)} \tag{4.7}$$

A iteração é repetida até estabilidade, isto é, até que duas iterações consecutivas produzam o mesmo resultado.

Em `morph.py`: `mm.infrec(f, g, b)` dilata f dentro de g até convergir.

A reconstrução morfológica é significativamente mais robusta que a abertura convencional porque filtra estruturas indesejadas **sem alterar a morfologia** dos objetos preservados. Enquanto a abertura suaviza cantos e deforma contornos devido à aplicação irrestrita do elemento estruturante, a reconstrução geodésica resgata os limites exatos dos objetos que interceptam o marcador, conforme ilustrado na Figure 4.7 com o elemento estruturante em cruz da Figure 4.6.

```
B_cruz = mm.secross()
mm.drawImagePlt(B_cruz, scale=20)
```

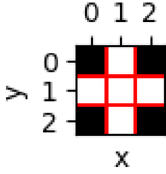


Figura 4.6: Elemento estruturante em formato de cruz (B_{cruz}) utilizado para conectividade-4.

```
# 1. Definição da Máscara (g): Imagem 10x10 com dois objetos isolados
g = np.array([
    [0,0,0,0,0,0,0,0,0,0],
    [0,1,1,1,0,0,0,1,1,0],
    [0,1,1,1,1,0,0,1,1,0],
    [0,1,1,1,1,0,0,0,0,0],
    [0,1,1,1,1,0,0,0,0,0],
    [0,1,1,1,0,0,0,0,0,0],
    [0,0,1,1,1,0,0,1,0,0],
    [0,0,1,1,1,0,0,1,1,0],
    [0,0,0,1,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0]], dtype=np.uint8) * 255

B_cruz = mm.secross()

# 2. Geração do Marcador (f)
f = mm.ero(g, mm.sebox())

# 3. Iterações da Dilatação Condicionada automatizadas em laço
iteracoes = []
titulos_iteracoes = []
img_atual = f.copy()

for i in range(1, 6):
    img_add = img_atual*80
    img_atual = mm.cdil(img_atual, g, B_cruz)
    iteracoes.append(img_add+img_atual)
    titulos_iteracoes.append(f"Dilatação Cond. (n={i})")

# Reconstrução Geodésica Final via biblioteca (ponto de controle)
img_reconstruida = mm.infrec(f, g, B_cruz)

# Verificação de convergência (o passo 5 deve ser idêntico à reconstrução final)
convergencia_ok = np.array_equal(img_reconstruida, iteracoes[-1])
print(f" Reconstrução alcançou estabilidade na iteração 5: {convergencia_ok}")

# 4. Exibição do pipeline evolutivo (Montagem dinâmica das listas)
mm.show(
    [g, f] + iteracoes + [img_reconstruida],
    titles=["Máscara (g)", "Marcador (f)"] + titulos_iteracoes + ["Reconstrução Final R_g(f)"],
    cols=8,
    figsize=(18, 3)
)
```

```
Reconstrução alcançou estabilidade na iteração 5: False
```

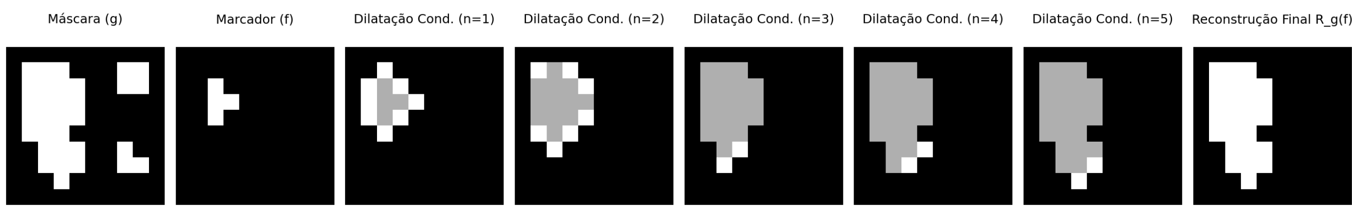


Figura 4.7: Pipeline de Reconstrução Morfológica por Dilatação Condicionada: a máscara contém dois objetos, o marcador isola apenas o núcleo do objeto principal, e as iterações subsequentes em laço reconstroem sua forma exata até a convergência.

4.3.4 Preenchimento de Buracos e Remoção de Bordas

Dois operadores baseados em reconstrução morfológica completam o *pipeline* de limpeza binária:

Preenchimento de buracos (`mm.clohole`) fecha qualquer buraco completamente cercado pelo objeto, independentemente do tamanho, sem alterar os contornos externos. O marcador é uma moldura (*frame*) — imagem com pixels ativos apenas na borda — aplicada sobre o complemento f^c:

$$\text{clohole}(f) = \left(R^{\delta}_{f^c}(\text{frame}(f^c))\right)^c \tag{4.8}$$

Remoção de objetos de borda (`mm.edgeoff`) elimina todos os objetos que tocam a borda da imagem, preservando apenas os completamente internos. O marcador é uma moldura aplicada sobre f:

$$\text{edgeoff}(f) = f \setminus R^{\delta}_{f}(\text{frame}(f)) \tag{4.9}$$

i Comparação dos dois operadores

Operador	Marcador	Máscara	Efeito
`mm.clohole`	frame de f^c	f^c	Preenche buracos internos
`mm.edgeoff`	frame de f	f	Remove objetos que tocam a borda

A evolução passo a passo dessas transformações geodésicas pode ser acompanhada nas figuras a seguir. A Figure 4.8 ilustra o mecanismo de inundação controlada do operador `mm.clohole`, evidenciando como a frente de onda preenche o plano de fundo externo e deixa isoladas apenas as cavidades internas. Em contrapartida, a Figure 4.9 detalha a dinâmica do operador `mm.edgeoff`, onde os objetos ancorados nas extremidades da matriz são integralmente mapeados a partir do *frame* limítrofe e, em seguida, eliminados por meio da subtração morfológica, restando exclusivamente os elementos totalmente contidos no domínio interno da imagem. A conectividade da propagação geodésica é controlada pelo elemento estruturante: `mm.sebox()` (vizinhança de 8) inclui componentes diagonais, enquanto `mm.secross()` (vizinhança de 4) os exclui — o que afeta quais objetos são atingidos pelo *frame* e, portanto, quais são preservados ou removidos.

```python
# Função auxiliar unificada para gerar o pipeline iterativo com realce visual
def gerar_pipeline_reconstrucao(marcador, mascara, kernel, passos=4, fator=80):
    iteracoes, titulos = [], []
    img_atual = marcador.copy()
    for i in range(1, passos + 1):
        img_visual = img_atual * fator
        img_atual = mm.cdil(img_atual, mascara, kernel)
        iteracoes.append(img_visual + img_atual)
        titulos.append(f"Iter. (n={i})")
    return iteracoes, titulos
```

```
# Imagem binária 10x10 com 3 cenários: objeto com buraco, objeto isolado e objeto na borda
f = np.array([
    [0,0,0,0,0,0,0,0,0,0],
    [0,1,1,1,1,0,0,0,0,1],
    [0,1,0,0,1,0,0,1,0,1],
    [0,1,0,0,1,0,0,1,0,1],
    [0,1,1,1,1,0,0,1,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,1,1,1,0,1,1,1,0],
    [0,0,1,1,1,0,1,0,1,0],
    [0,0,1,1,1,0,1,1,1,0],
    [0,0,0,0,0,0,0,0,0,1]], dtype=np.uint8) * 255

B_cruz = mm.secross()
f_c = mm.neg(f)              # Utiliza o operador de negação nativo da mm

#   PIPELINE CLOHOLE
# O marcador do clohole teórico é a borda externa
marcador_ch = mm.frame(f, border=1)
iters_ch, tits_ch = gerar_pipeline_reconstrucao(marcador_ch, f_c, B_cruz, passos=5)

# Resultado final calculado por extenso e validado com a função nativa mm.clohole
img_clohole = mm.neg(mm.infrec(marcador_ch, f_c, B_cruz))
print(f" Validação clohole: {np.array_equal(img_clohole, mm.clohole(f))}")

mm.show(
    [f, marcador_ch] + iters_ch + [img_clohole],
    titles=["f original", "Marcador (Borda)"] + tits_ch + ["clohole(f)"],
    cols=8, figsize=(18, 3), axis=True
)
```

```
Validação clohole: True
```

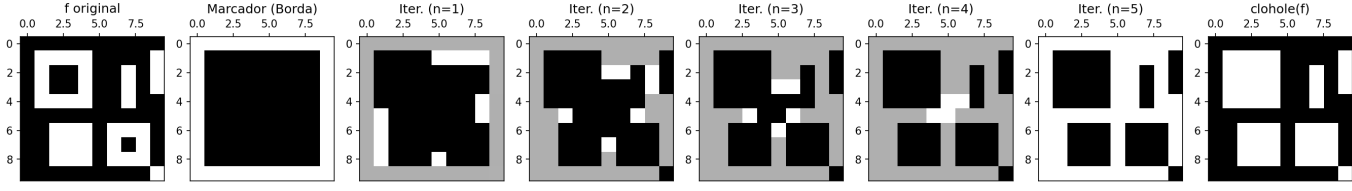


Figura 4.8: Pipeline de preenchimento de buracos (*clohole*): o marcador é gerado restringindo o frame da borda ao complemento f^c. As iterações de dilatação condicionada preenchem o fundo externo e o operador final preserva unicamente os buracos internos isolados.

```
#   PIPELINE EDGEOFF
B_box = mm.sebox()
marcador_eo = marcador_ch & f
iters_eo, tits_eo = gerar_pipeline_reconstrucao(marcador_eo, f, B_box, passos=5)

# Resultado final por definição e validação nativa
img_edgeoff =  mm.edgeoff(f, B_box)
print(f" Validação edgeoff: {np.array_equal(img_edgeoff, mm.edgeoff(f))}")

mm.show(
    [f, marcador_eo] + iters_eo + [img_edgeoff],
    titles=["f original", "Marcador (Borda)"] + tits_eo + ["edgeoff(f)"],
    cols=8, figsize=(18, 3)
)
```

```
Validação edgeoff: True
```

Figura 4.9: Pipeline de eliminação de estruturas de borda (*edgeoff*): o marcador captura as raízes conectadas às extremidades da matriz, a reconstrução delimita a extensão desses elementos e a subtração booleana preserva unicamente os objetos totalmente internos.

4.3.5 Pipeline de Limpeza Binária com CLAHE

Com base na análise anterior — em que o CLAHE fornece a melhor separação bimodal ($\sigma_B^2 \approx 2{,}47 \times 10^3$) e o operador `mm.clohole` preenche satisfatoriamente as cavidades internas das moedas —, o *pipeline* final de segmentação, ilustrado na Figure 4.10, é estruturado pelo seguinte fluxo computacional:

$$\text{gray} \xrightarrow{\text{CLAHE}} \xrightarrow{\text{Otsu}} \xrightarrow{\text{open}} \xrightarrow{\text{clohole}} \xrightarrow{\text{open}} \xrightarrow{\text{edgeoff}} \text{segmentação}$$

Após a etapa do `mm.clohole`, aplica-se uma abertura morfológica com um elemento estruturante de dimensões amplas (`mm.sedisk(33)`, correspondente a um disco de raio 33). Essa operação visa eliminar pequenos artefatos e regiões residuais, e resíduos de fundo que o processo de preenchimento geodésico pode preencher artefatos de segmentação no fundo que ficaram completamente cercados após a abertura inicial, não necessariamente ligados às fronteiras. Observa-se também que, devido à presença de moedas tangenciando, mas não tocando os limites da matriz, o operador `mm.edgeoff` subsequente não remove nenhuma moeda, fazendo com que a imagem final equivalha ao resultado da abertura restrito aos objetos estritamente internos.

Por que a abertura após o clohole?

O operador `mm.clohole` preenche **todos** os buracos fechados, incluindo eventuais artefatos de segmentação no plano de fundo que tenham sido isolados acidentalmente. A abertura morfológica (erosão seguida de dilatação) remove eficientemente esses objetos espúrios menores que o contorno do *kernel*, preservando as moedas genuínas devido à sua propriedade de filtragem geométrica por inclusão.

```
#   Pré-processamento: apenas CLAHE
img_clahe0 = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8)).apply(img_coins_gray)

#   Etapa 1: Binarização Otsu
img_bin = mm.threshold(img_clahe0)

#   Etapa 2: Abertura - remove ruídos brancos de fundo
img_open = mm.open(img_bin, mm.sedisk(9))

#   Etapa 3: clohole - fecha todos os buracos internos
img_hole = mm.clohole(img_open)

#   Etapa 4: Abertura (kernel grande) - remove artefatos do clohole
img_limpo = mm.open(img_hole, mm.sedisk(33))

#   Etapa 5: edgeoff - remove objetos que tocam a borda
img_final = mm.edgeoff(img_limpo)

mm.show(
```

```
    [img_clahe0,  img_bin,                img_open,
     img_hole,    img_limpo,              img_final],
    titles=["CLAHE",            "Otsu",                "Abertura (r=9)",
            "clohole",          "Abertura (r=33)",     "Final (edgeoff)"],
    cols=6, figsize=(18, 6)
)
```

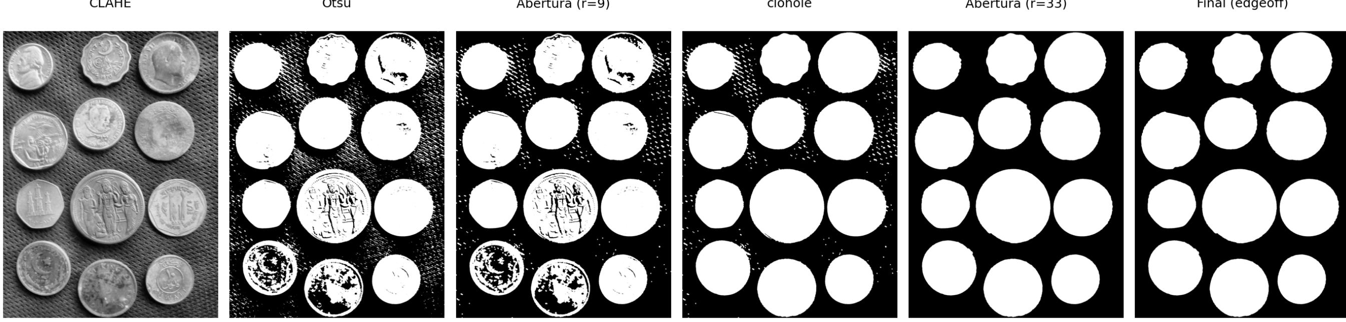


Figura 4.10: Pipeline completo de segmentação com CLAHE: Otsu → fechamento → clohole → abertura (remoção de ruído) → edgeoff.

4.3.6 Morfologia em Tons de Cinza

Os operadores morfológicos estendem-se para imagens em tons de cinza substituindo as operações de conjunto por **mínimo** (erosão) e **máximo** (dilatação), conforme definido nas Equation 4.3 e Equation 4.4. Para elemento estruturante plano ($b \equiv 0$), erosão reduz-se ao **mínimo local** e dilatação ao **máximo local**.

Três operadores derivados são especialmente úteis:

Gradiente morfológico — destaca bordas como a diferença entre dilatação e erosão:

$$\text{grad}_B(f) = (f \oplus B) - (f \ominus B) \tag{4.10}$$

Top-hat — destaca estruturas brilhantes menores que o elemento estruturante (diferença entre f e sua abertura):

$$\text{top-hat}_B(f) = f - (f \circ B) \tag{4.11}$$

Black-hat — destaca estruturas escuras menores que o elemento estruturante (diferença entre o fechamento e f):

$$\text{black-hat}_B(f) = (f \bullet B) - f \tag{4.12}$$

Top-hat extrai elementos brilhantes que são menores que o elemento estruturante B, enquanto o *Black-hat* extrai elementos escuros menores que B.

Os três operadores, ilustrados na Figure 4.11, estão disponíveis em `mm.gradm`, `mm.tophat` e `mm.blackhat`.

```
import io
import matplotlib.pyplot as plt
import numpy as np

def fig2img(fig):
    b = io.BytesIO(); fig.savefig(b, format='png', dpi=100); plt.close(fig); b.seek(0)
    return (plt.imread(b)[:, :, :3] * 255).astype(np.uint8)

def plot_hist(img, title):
    fig, ax = plt.subplots(figsize=(4, 3))
    h = mm.hist(img)
    # CORREÇÃO: range usa len(h) dinamicamente para casar com o retorno da biblioteca
    ax.bar(range(len(h)), h, color='steelblue', width=1, edgecolor='steelblue')
```

```
    ax.set(title=title, xlim=(0, 255)); plt.tight_layout()
    return fig2img(fig)

# 1. Processamento morfológico base
B = mm.sedisk(19)
operadores = [
    ("Original", img_coins_gray),
    ("Erosão", mm.ero(img_coins_gray, B)),
    ("Dilatação", mm.dil(img_coins_gray, B)),
    ("Gradiente Morf.", mm.gradm(img_coins_gray, B)),
    ("Top-hat", mm.tophat(img_coins_gray, B)),
    ("Black-hat", mm.blackhat(img_coins_gray, B))
]

# 2. Montagem dinâmica do par: [Imagem, Histograma]
imgs, titles = [], []
for nome, img in operadores:
    imgs += [img, plot_hist(img, f"Hist. {nome}")]
    titles += [nome, f"Hist. {nome}"]

# 3. Exibição final em grade de duas colunas (Imagem | Histograma)
mm.show(imgs, titles=titles, cols=4, figsize=(10, 8))
```

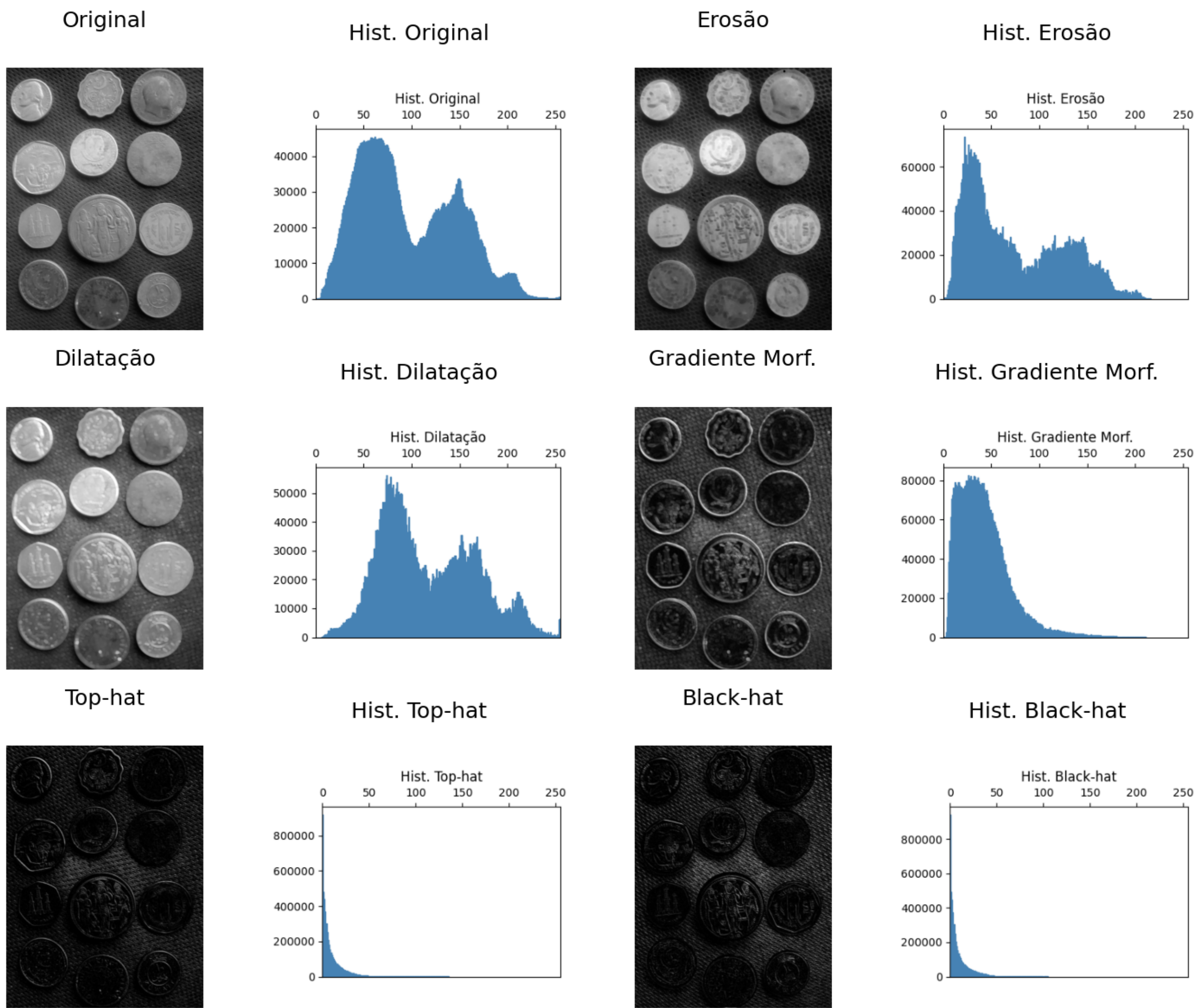


Figura 4.11: Morfologia em tons de cinza emparelhada com seus respectivos histogramas: imagem seguida por sua distribuição de frequências de intensidade. Elemento estruturante: disco de raio 19.

Os operadores morfológicos apresentados anteriormente serão agora utilizados como ferramentas de refinamento e geração de marcadores para métodos de segmentação mais avançados, apresentados a seguir.

4.4 Segmentação de Imagens: Fundamentação e Taxonomia

A **segmentação de imagens** consiste em dividir a imagem em regiões associadas a objetos ou estruturas de interesse. Em PDI, ela representa a transição entre o processamento de baixo nível — como filtragem e realce — e etapas de análise mais avançadas, como extração de características, reconhecimento e interpretação da cena.

Formalmente, o objetivo da segmentação consiste em decompor o domínio espacial completo de uma imagem, denotado por Ω, em uma partição de subconjuntos $\{R_1, R_2, \dots, R_n\}$ que satisfaça simultaneamente os critérios de **completeza** e **disjunção**:

$$\bigcup_{i=1}^{n} R_i = \Omega, \quad \text{onde} \quad R_i \cap R_j = \emptyset \quad \forall \quad i \neq j \tag{4.13}$$

Cada sub-região R_i deve constituir um domínio **homogêneo** segundo um predicado de similaridade definido

sobre propriedades locais — intensidade, cor ou textura — e, concomitantemente, ser **distinta** em relação às regiões adjacentes. As abordagens para a obtenção dessa partição organizam-se em três famílias, fundamentadas em critérios de continuidade, conectividade e similaridade estatística, conforme sintetizado na Table 4.5.

Tabela 4.5: Taxonomia dos métodos de segmentação baseados em propriedades de homogeneidade e conectividade espacial.

Abordagem	Critério Topológico-Espacial	Operadores de Referência
Limiarização	Descontinuidade e particionamento do espaço de intensidades	Critério de Otsu, Limiarização Global e Local
Baseada em Regiões	Homogeneidade local e busca por conectividade	Crescimento de Regiões, *Watershed* por Inundação
Agrupamento (*Clustering*)	Similaridade e otimização em espaços de características	Algoritmo *k-means*, Modelos de Mistura Gaussiana (GMM)

Até este ponto do capítulo, o desenvolvimento prático abordou a *Limiarização* — mediante o acoplamento entre equalização adaptativa CLAHE e o limiar global de Otsu — estendida por operadores de *Crescimento de Regiões* aplicados ao confinamento geodésico, materializados pelas funções `mm.infrec`, `mm.clohole` e `mm.edgeoff`. O resultado desse *pipeline* é uma máscara binária livre de ruídos isolados.

Contudo, em cenários onde os objetos segmentados encontram-se sobrepostos ou vinculados por junções estreitas de pixels, técnicas baseadas em limiarização são insuficientes para satisfazer a condição de disjunção imposta pela Equation 4.13. Para resolver essa limitação topológica, as próximas seções introduzem três operadores que atuam em sinergia morfológica: a **Rotulação**, a **Transformada de Distância** e o algoritmo de segmentação por ***Watershed*** baseado em marcadores.

4.4.1 Rotulação

A **rotulação de componentes conexas** (*connected component labeling*) é o operador que atribui um identificador inteiro e único a cada conjunto de pixels pertencentes à mesma componente conexa em uma imagem binária. Formalmente:

> Dada uma imagem binária f e uma conectividade $\mathcal{C}$ (4 ou 8 vizinhos), o algoritmo de rotulação retorna uma imagem g tal que todos os pixels pertencentes à mesma componente conexa recebem o mesmo inteiro positivo distinto, e pixels de componentes distintas recebem inteiros diferentes.

A conectividade define quais pixels são considerados vizinhos diretos de um pixel (x, y):

- **Conectividade-4:** apenas os 4 vizinhos ortogonais (norte, sul, leste, oeste).
- **Conectividade-8:** os 4 ortogonais mais os 4 diagonais, totalizando 8 vizinhos.

A escolha da conectividade afeta diretamente quais regiões são fundidas em uma mesma componente, como ilustrado na Figure 4.13. Ver outro exemplo no simulador da Figure 4.12.

Algoritmo (*flood-fill* com pilha):

1. Criar a imagem de saída g inicializada com zeros, de mesma dimensão que f.
2. Inicializar um contador $c \leftarrow 1$.
3. Varrer f em **ordem *raster*** (linha a linha, da esquerda para a direita) até localizar um pixel ativo ($f[x, y] \neq 0$) ainda não rotulado ($g[x, y] = 0$).
4. Inserir esse pixel em uma pilha $\mathcal{P}$, inicialmente vazia.
5. Enquanto $\mathcal{P} \neq \emptyset$:
   - Desempilhar o pixel (i, j) e atribuir $g[i, j] \leftarrow c$.
   - Empilhar os vizinhos ativos ainda não visitados de (i, j) segundo $\mathcal{C}$ que sejam ativos em f e ainda não rotulados em g.

6. Incrementar $c \leftarrow c + 1$ e retornar ao passo 3.

A implementação em `morph.py` expõe duas versões: `label0`, que reproduz o algoritmo acima explicitamente para fins didáticos, e `label`, que delega ao `cv2.connectedComponents` otimizado:

```
@staticmethod
def label(f):
    _, lbl = cv2.connectedComponents(f); return lbl

@staticmethod
def label0(f, b=np.ones((3,3), dtype='uint8')):
    """Rotulagem por flood-fill com pilha."""
    h, w = f.shape
    g = np.zeros(f.shape, dtype=int)
    cor = 1
    for x in range(h):
        for y in range(w):
            if f[x,y] and not g[x,y]:
                pilha = [[x,y]]
                while pilha:
                    i,j = pilha.pop(); g[i,j] = cor
                    for vy,vx,bv in mm._viz(f,b,i,j):
                        if bv and f[vy,vx] and not g[vy,vx]:
                            pilha.append([vy,vx])
                cor += 1
    return g
```

O auxiliar `_viz` itera sobre a janela estruturante `b` centrada em (i, j), gerando somente os vizinhos válidos dentro dos limites da imagem — a conectividade desejada é inteiramente determinada pela forma de `b` passada ao algoritmo.

Exemplo didático — efeito da conectividade:

```
# Imagem binária 10×10 com componentes diagonalmente adjacentes
f = np.array([
    [0,0,0,0,0,0,0,0,0,0],
    [0,1,0,0,0,0,0,0,0,0],
    [0,0,1,0,0,0,0,0,0,0],  # diagonal com linha anterior
    [0,0,0,1,0,0,1,1,0,0],
    [0,0,0,0,0,0,1,1,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,1,1,1,0,0,0,0,0,0],
    [0,1,0,1,0,0,0,1,0,0],
    [0,1,1,1,0,0,0,0,1,0],  # diagonal com linha anterior
    [0,0,0,0,0,0,0,0,0,0]], dtype=np.uint8) * 255

# Elemento estruturante cruz (conectividade-4) e quadrado (conectividade-8)
B4 = mm.secross()                        # conectividade-4
B8 = np.ones((3,3), dtype=np.uint8)     # conectividade-8

# Rotulação com label0 (didático) e label (cv2)
lbl4_didatico = mm.label0(f, B4)
lbl8_didatico = mm.label0(f, B8)

_, lbl4_cv2 = cv2.connectedComponents(f, connectivity=4)
_, lbl8_cv2 = cv2.connectedComponents(f, connectivity=8)

# Validação cruzada
print(f" label0 (C4) == cv2 (C4): {np.array_equal(lbl4_didatico, lbl4_cv2)}")
print(f" label0 (C8) == cv2 (C8): {np.array_equal(lbl8_didatico, lbl8_cv2)}")
print(f"   Componentes C4: {lbl4_cv2.max()}  |  Componentes C8: {lbl8_cv2.max()}")
```

Figura 4.12: Simulador interativo de rotulação de componentes conexas (*connected component labeling*): visualização da expansão flood-fill, conectividade-4 e conectividade-8.

```
# Normalização para visualização
def norm_label(lbl):
    out = np.zeros_like(lbl, dtype=np.uint8)
    for i, v in enumerate(np.unique(lbl)[1:], 1):
        out[lbl == v] = int(i * 255 / lbl.max())
    return out

mm.show(
    [f, norm_label(lbl4_cv2), norm_label(lbl8_cv2)],
    titles=[
        "f original",
        f"Rotulação C4\n({lbl4_cv2.max()} componentes)",
        f"Rotulação C8\n({lbl8_cv2.max()} componentes)"
    ],
    cols=3, figsize=(12, 4), axis=True
)
```

```
label0 (C4) == cv2 (C4): True
label0 (C8) == cv2 (C8): True
 Componentes C4: 7  |  Componentes C8: 4
```

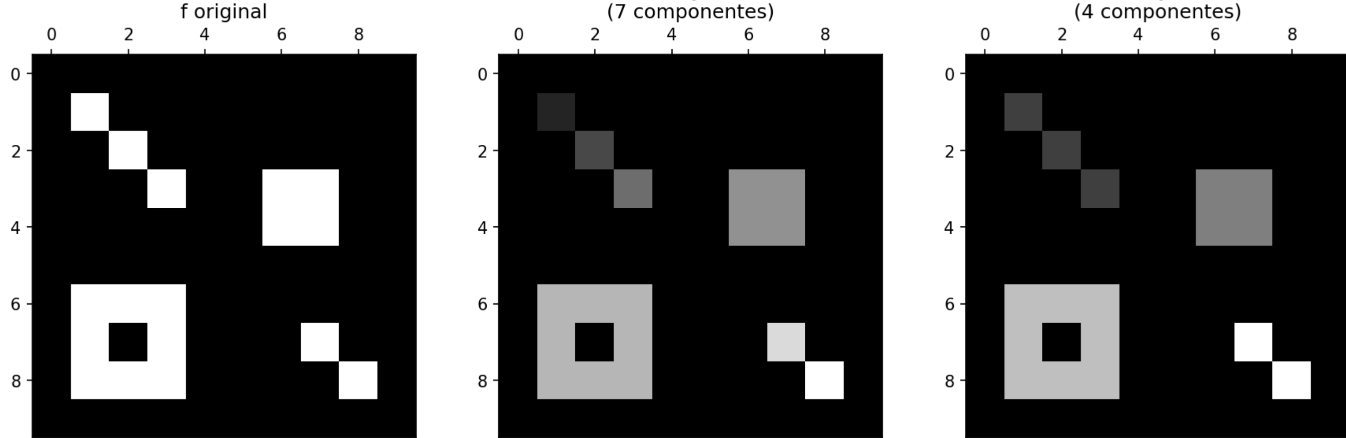


Figura 4.13: Efeito da conectividade na rotulação: a imagem binária 10×10 contém pixels diagonalmente adjacentes. Com conectividade-4, esses pixels formam componentes distintas; com conectividade-8, fundem-se em uma única componente.

4.4.2 Transformada de Distância

A **Transformada de Distância** (TD) é um operador que, aplicado a uma imagem binária f, produz uma imagem em níveis de cinza D na qual o valor de cada pixel ativo representa sua distância ao pixel de fundo mais próximo:

$$D(x, y) = \min_{(x',y')\,:\,f(x',y')=0} d\big((x, y),\, (x', y')\big)$$

onde $d(\cdot, \cdot)$ é uma métrica de distância — tipicamente Euclidiana (L_2). O resultado é uma representação topográfica dos objetos: pixels no interior profundo assumem valores elevados, enquanto pixels próximos à borda dos objetos apresentam baixos valores de distância. Os **máximos locais** de D correspondem aos centros geométricos aproximados de cada objeto — propriedade relevante para a geração de marcadores no *watershed*.

A TD admite ainda uma interpretação morfológica iterativa: aplicar erosões sucessivas com uma função estruturante b especial até que o objeto se extinga equivale a atribuir a cada pixel o resultado de erosões

sucessivas, segundo o elemento estruturante adotado. Essa visão é materializada em `mm.dist1()`, enquanto `mm.dist()` delega ao operador L_2 otimizado do OpenCV:

```
@staticmethod
def dist(f):
    y = cv2.distanceTransform(f, cv2.DIST_L2, 5)
    return y.astype('uint8') if y.max() <= 255 else y.astype('uint16')

@staticmethod
def dist1(f, b):
    """TD iterativa por erosões sucessivas."""
    g = f.copy()
    while True:
        f = g.copy(); g = mm.ero1(g, b)
        if np.array_equal(f, g): break
    return g
```

`dist1` é funcionalmente equivalente à TD na métrica induzida por `b`: com o elemento cruz (`secross`), a métrica resultante é a ***Manhattan*** (L_1); com o quadrado (`sebox`), aproxima-se da distância de ***Chebyshev*** (L_∞). Por ser iterativa com varredura completa a cada passo, `dist1` é impraticável em imagens grandes — seu uso é restrito a fins didáticos e verificação em imagens pequenas.

A Figure 4.14 apresenta um simulador interativo da TD: dado um pixel do objeto, é possível mover o cursor sobre a imagem binária e observar, em tempo real, o valor da TD naquela posição — isto é, a TD ao pixel de fundo mais próximo. A Figure 4.15 apresenta um exemplo prático em Python da TD.

Exemplo didático — TD iterativa vs. direta em imagem 10×10:

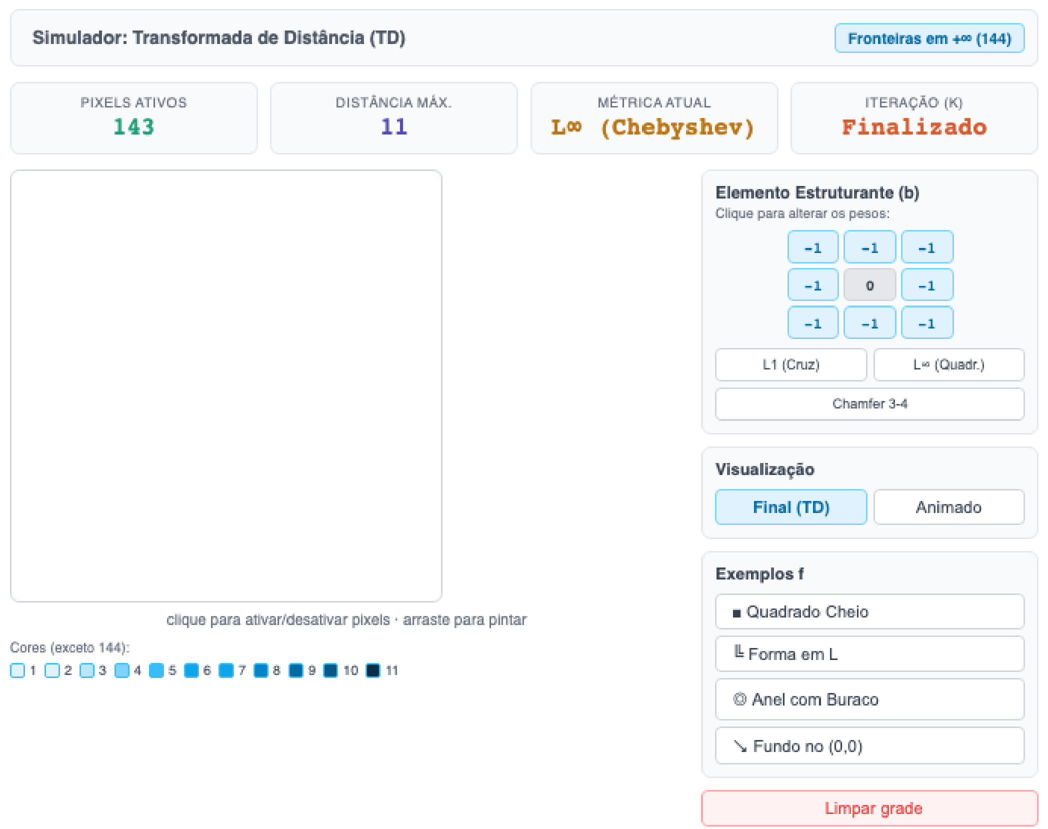
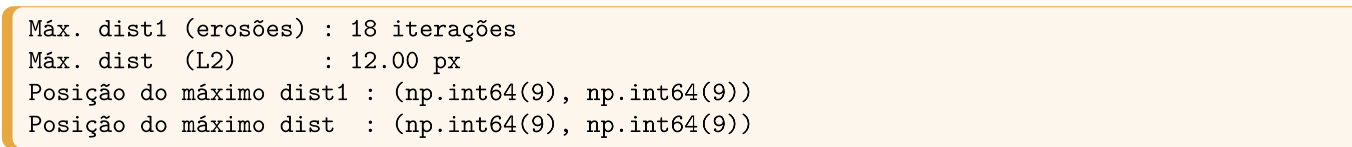


Figura 4.14: Simulador interativo da Transformada de Distância (TD) iterativa via erosão em tons de cinza. Pixels fora da imagem agora assumem o valor máximo (144), propagando os custos apenas a partir do fundo interno.

```python
import numpy as np

# Imagem binária 10×10 com um único pixel de fundo (0,0) e o resto como objeto (255)
f = np.ones((10,10), dtype=np.uint8) * 255
f[0,0] = 0

B_cruz = np.array([
    [-np.inf, -1, -np.inf],
    [-1,       0, -1],
    [-np.inf, -1, -np.inf]
], dtype=float)

d_iter = mm.dist1(f, B_cruz)
d_l2   = mm.dist(f)

print(f"Máx. dist1 (erosões) : {d_iter.max()} iterações")
print(f"Máx. dist  (L2)      : {d_l2.max():.2f} px")
print(f"Posição do máximo dist1 : {np.unravel_index(d_iter.argmax(), d_iter.shape)}")
print(f"Posição do máximo dist  : {np.unravel_index(d_l2.argmax(),  d_l2.shape)}")

mm.show(
    [f,         d_iter,                     d_l2],
    titles=["f original", "mm.dist1\n(erosões com cruz)", "mm.dist\n(L2 Euclidiana)"],
    cols=3, figsize=(12, 4), axis=True
)
```



```
Máx. dist1 (erosões) : 18 iterações
Máx. dist  (L2)      : 12.00 px
Posição do máximo dist1 : (np.int64(9), np.int64(9))
Posição do máximo dist  : (np.int64(9), np.int64(9))
```

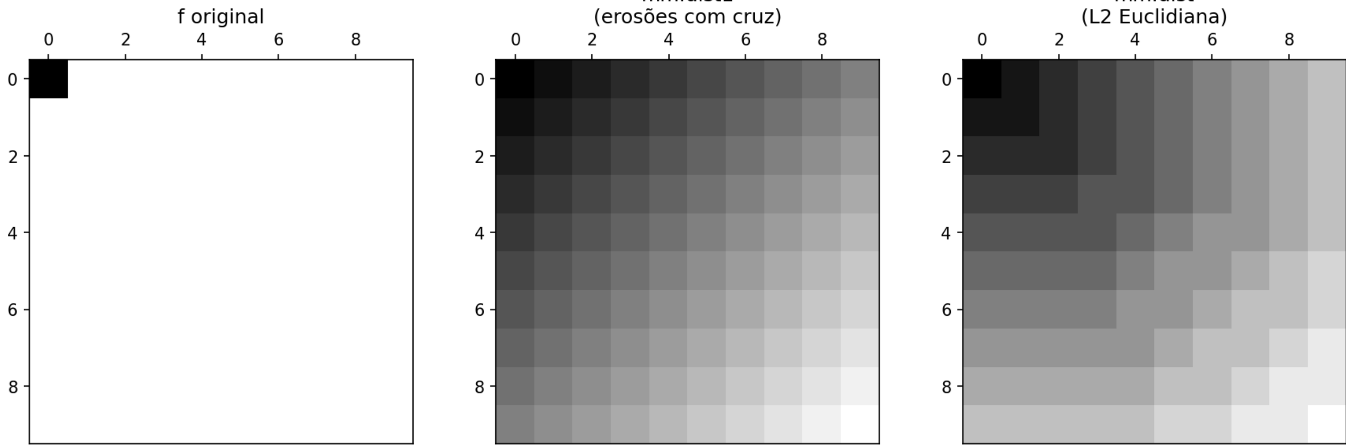


Figura 4.15: Transformada de Distância em imagem binária 10×10. Esquerda: imagem original (foreground = 255). Centro: mm.dist1 iterativa (erosões com elemento cruz), exibindo o número de erosões sobrevividas por cada pixel. Direita: mm.dist (L2 Euclidiana). Os valores internos confirmam a equivalência entre as duas abordagens na métrica induzida pelo elemento estruturante.

A anotação dos valores numéricos diretamente sobre os pixels permite verificar que `dist1` atribui a cada pixel exatamente o número de erosões necessárias para eliminá-lo — confirmando a correspondência com a distância *Manhattan* induzida pelo elemento cruz. Os máximos coincidem geometricamente com o centro do objeto em ambas as versões, validando o uso de `mm.dist` como substituto eficiente de `mm.dist1` em imagens de escala real.

4.4.3 Segmentação por *Watershed*

O algoritmo ***Watershed*** baseado em TD, regiões centrais dos objetos correspondem aos máximos topográficos, enquanto regiões de separação atuam como divisores entre bacias. O processo simula uma inundação progressiva a partir de **marcadores** (sementes pré-definidas que identificam com certeza o interior de cada objeto). Quando a "água" de duas bacias distintas se encontra, ergue-se uma represa: a *watershed line*. Essa linha delimita objetos sobrepostos ou em contato, solucionando cenários em que a limiarização global falha em separá-los.

A implementação didática explora o conceito de crescimento de regiões (*region growing*):

```
def watershed0(f, b=np.zeros((3,3), dtype='uint8'), op='region'):
    """Watershed didático por propagação de rótulos."""
    f = mm.label0(f, b); g = f.copy()
    while g.min() == 0:
        for x in range(f.shape[0]):
            for y in range(f.shape[1]):
                for vy, vx, _ in mm._viz(f, b, x, y):
                    if g[x,y] == 0 and g[x,y] < f[vy,vx]:
                        g[x,y] = f[vy,vx]
        f = g.copy()
    return g if op == 'region' else mm.gradm(g, mm.secross())

def watershed(f, mark, op='region'):
    mark = mark*255 if mark.max()==1 else mark
    if len(f):
        _, markers = cv2.connectedComponents(mark)
        w = cv2.watershed(f, markers)
        if op=='line': f[markers==-1]=[255,0,0]; return f
        return w
    from scipy import ndimage as ndi
    from skimage.segmentation import watershed
    fones = np.ones_like(mark)*255
    w = watershed(fones, ndi.label(mark)[0], mask=fones)
    if op=='line':
        return np.array((w-cv2.erode(w.astype('uint16'),mm.sebox()))>0,dtype='uint16')
    return w
```

A função `watershed0` ilustra o princípio da colisão de bacias propagando rótulos puramente por proximidade espacial (uma aproximação plana semelhante a um Diagrama de Voronoi). Já o `cv2.watershed` implementa o verdadeiro modelo topográfico: ele inunda a imagem respeitando os valores de intensidade/gradiente dos pixels, garantindo que as linhas de fronteira se formem exatamente sobre as cristas do relevo.

O *pipeline* morfológico típico do *watershed* baseado em marcadores segue as etapas resumidas na Table 4.6.

Tabela 4.6: Pipeline morfológico típico do *watershed* baseado em marcadores.

Etapa	Operação	Finalidade
1	Limiarização	Separação inicial objeto/fundo
2	Abertura/Fechamento	Remoção de ruído e buracos
3	Dilatação da Máscara	Identificar "Fundo Certo"
4	Transf. Distância + Limiar	Identificar "Frente Certa" (Marcadores)
5	Região Incerta	Fundo Certo − Frente Certa
6	cv2.watershed	Propagar marcadores pela região incerta

A Figure 4.16 apresenta um simulador iterativo que ilustra essa dinâmica: os marcadores comportam-se

como fontes de inundação que se expandem progressivamente pela máscara correspondente à região incerta. Quando duas bacias tentam ocupar simultaneamente o mesmo pixel, esse ponto passa a ser classificado como fronteira (`-1`). A Figure 4.17 apresenta um exemplo completo do algoritmo *watershed* em Python utilizando `morph.py`, enquanto a Figure 4.18 ilustra sua aplicação na separação de moedas adjacentes em uma imagem binária.

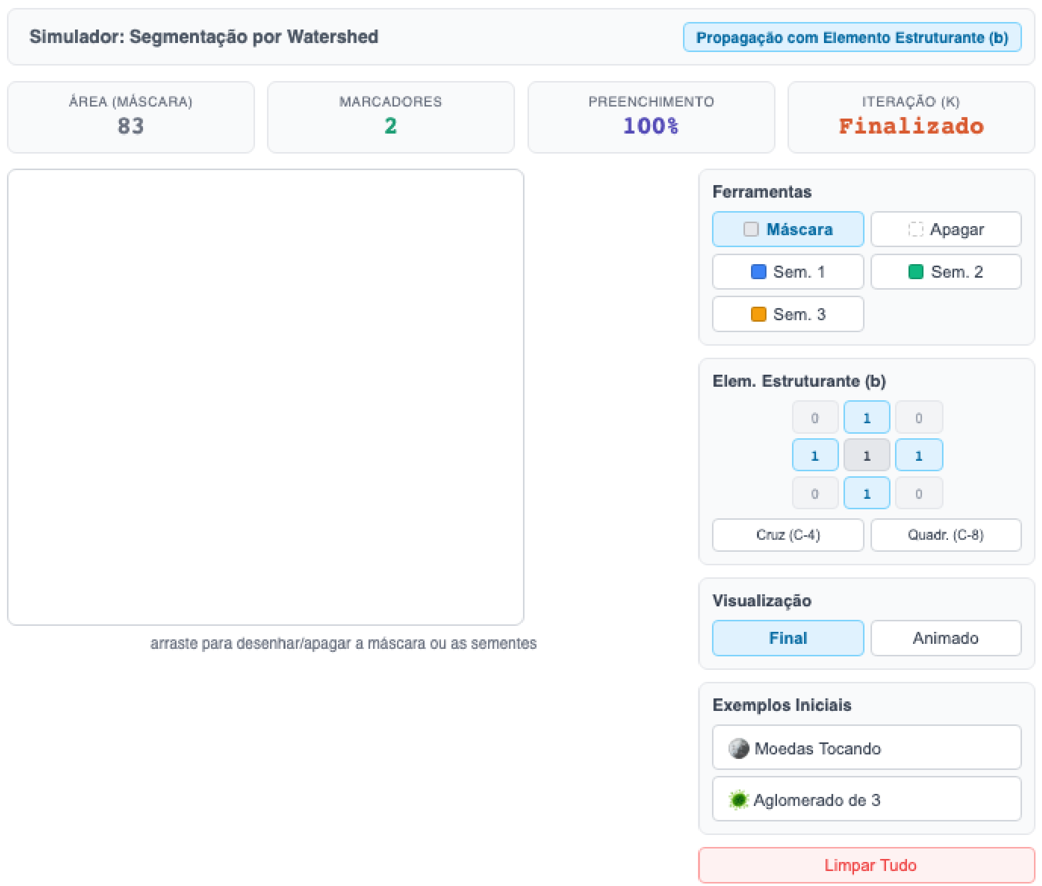


Figura 4.16: Simulador interativo do Algoritmo Watershed por propagação morfológica. Desenhe a máscara, posicione os marcadores e ajuste o Elemento Estruturante para observar a inundação. Quando as bacias se encontram simultaneamente, o empate é resolvido assumindo uma das regiões de forma aleatória.

```
# Imagem sintética: dois discos sobrepostos
f_sint = np.zeros((20, 20), dtype=np.uint8)
cv2.circle(f_sint, (6,  10), 5, 255, -1)
cv2.circle(f_sint, (14, 10), 5, 255, -1)

B8 = np.ones((3, 3), dtype=np.uint8)

#   Marcadores: máximo da transformada de distância
dist     = mm.dist(f_sint)
dist_vis = cv2.normalize(dist.astype(np.float32), None, 0, 255,
                         cv2.NORM_MINMAX).astype(np.uint8)

# os marcadores são os picos mais altos da topografia da TD,
# representando o centro "seguro" de cada objeto
_, fg    = cv2.threshold(dist.astype(np.float32), 0.8 * dist.max(), 255, 0)
fg       = fg.astype(np.uint8)

bg       = cv2.dilate(f_sint, B8, iterations=3)
unknown = cv2.subtract(bg, fg)
```

```
n_markers, markers = cv2.connectedComponents(fg)
markers = markers + 1
markers[unknown == 255] = 0

#   watershed (cv2)
f_bgr      = cv2.cvtColor(f_sint, cv2.COLOR_GRAY2BGR)
markers_ws = markers.copy()
cv2.watershed(f_bgr, markers_ws)

ws_region = cv2.normalize(
    np.where(markers_ws > 1, markers_ws, 0).astype(np.float32),
    None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)
ws_line = np.where(markers_ws == -1, 255, 0).astype(np.uint8)

mm.show(
    [f_sint, dist_vis, fg, ws_region, ws_line],
    titles=[
        "f sintética", "mm.dist ($L_2$)", "Frente (marcadores)",
        "watershed (cv2)\nop='region'",   "watershed (cv2)\nop='line'"
    ],
    cols=5, figsize=(16, 4)
)
```

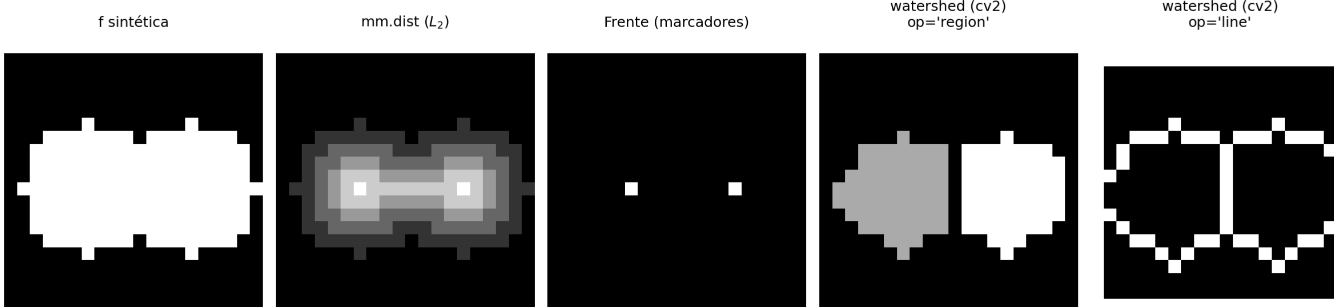


Figura 4.17: Pipeline watershed (cv2) em imagem binária 20×20 com dois discos sobrepostos: transformada de distância, marcadores por threshold e resultado final.

Aplicação em moedas sobrepostas:

4.5 Extração de Componentes e Descritores de Forma

Após segmentação e refinamento morfológico, o próximo passo é **rotular** cada região conexa e extrair suas propriedades. Um **componente conexo** é um conjunto máximo de pixels brancos mutuamente conectados (4- ou 8-conectividade).

Duas funções do OpenCV abordam esse problema por ângulos complementares:

	`connectedComponentsWithStats`	`findContours`
Retorna	rótulo por pixel + estatísticas	sequência de pontos da borda
Descritores diretos	área, bounding box, centróide	perímetro, forma, hierarquia
Objetos em contato	tende a fundir regiões conectadas	tende a retornar um único contorno externo
Uso típico	contar, filtrar, colorir regiões	análise de forma, polígonos

4.5.1 connectedComponentsWithStats

A Figure 4.19 demonstra a extração na imagem de moedas.

```
img_base = img_coins_gray

# Dilatar img_final para conectar moedas (simula sobreposição real)
img_sobrepostas = mm.dil(img_final, mm.sebox(40))

#   Etapas 1-2
kernel  = mm.sebox(2)
opening = mm.open(opening, kernel)

#   Etapas 2-4
bg      = cv2.dilate(opening, kernel, iterations=3)
dist    = mm.dist(opening)
dist_vis = cv2.normalize(dist.astype(np.float32), None, 0, 255,
                         cv2.NORM_MINMAX).astype(np.uint8)

_, fg   = cv2.threshold(dist.astype(np.float32), 0.5 * dist.max(), 255, 0)
fg      = fg.astype(np.uint8)
unknown = cv2.subtract(bg, fg)

#   Etapas 5-6
n_markers, markers = cv2.connectedComponents(fg)
markers = markers + 1
markers[unknown == 255] = 0

img_bgr = cv2.cvtColor(img_base, cv2.COLOR_GRAY2BGR)
ws_line = mm.watershed(img_bgr.copy(), fg, op='line')

# Imagem anotada
markers_ann = markers.copy()
img_annotated = cv2.cvtColor(img_base, cv2.COLOR_GRAY2BGR)
cv2.watershed(img_annotated, markers_ann)

for m in range(2, n_markers + 1):
    mask = (markers_ann == m).astype(np.uint8)
    if mask.sum() < 500: continue
    cx = int(np.mean(np.where(mask)[1]))
    cy = int(np.mean(np.where(mask)[0]))
    cv2.putText(img_annotated, str(m - 1), (cx - 25, cy + 20),
                cv2.FONT_HERSHEY_SIMPLEX, 3.2, (0, 255, 0), 8, cv2.LINE_AA)

mask_ann = (markers_ann == -1).astype(np.uint8)
mask_ann = cv2.dilate(mask_ann, np.ones((11, 11), np.uint8), iterations=2)
img_annotated[mask_ann == 1] = [0, 0, 255]
img_annotated = cv2.cvtColor(img_annotated, cv2.COLOR_BGR2RGB)
ws_line_rgb   = cv2.cvtColor(ws_line, cv2.COLOR_BGR2RGB)

print(f"Objetos detectados: {n_markers - 1}")

mm.show(
    [img_base, img_sobrepostas, dist_vis, fg, ws_line_rgb, img_annotated],
    titles=[
        "Original", "img_final dilatada\n(sobrepostas)",
        "mm.dist ($L_2$)", "Frente (marcadores)",
        "mm.watershed\nop='line'", "Anotado"
    ],
    cols=3, rows=2, figsize=(12, 8)
)
```

```
NameError: name 'opening' is not defined
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[20], line 13
      9 img_sobrepostas = mm.dil(img_final, mm.sebox(40))
     10
```

```
# 1. Rotulagem e estatísticas
n, labels, stats, centroids = cv2.connectedComponentsWithStats(
    img_final, connectivity=8
)

# 2. Coloração dos componentes
np.random.seed(4)
colors    = np.random.randint(50, 255, (n, 3), dtype=np.uint8)
colors[0] = [0, 0, 0]                              # fundo preto
img_colored   = colors[labels]
img_annotated = img_colored.copy()

# 3. Tabela de descritores
print(f"Componentes detectados (excluindo fundo): {n - 1}")
print(f"\n{'ID':>4} {'Área':>8} {'cx':>6} {'cy':>6} {'w':>6} {'h':>6}")
print("-" * 42)
for i in range(1, n):
    area = stats[i, cv2.CC_STAT_AREA]
    cx, cy = int(centroids[i, 0]), int(centroids[i, 1])
    w, h   = stats[i, cv2.CC_STAT_WIDTH], stats[i, cv2.CC_STAT_HEIGHT]
    print(f"{i:>4} {area:>8} {cx:>6} {cy:>6} {w:>6} {h:>6}")
    for cor, esp in [((0,0,0), 10), ((255,0,0), 5)]:
        cv2.putText(img_annotated, f"{i}: {area}",
                    (cx - 150, cy + 18),
                    cv2.FONT_HERSHEY_SIMPLEX, 2.0, cor, esp, cv2.LINE_AA)

mm.show(
    [img_coins_gray, img_final, img_colored, img_annotated],
    titles=["Original", "Segmentação Final", "Componentes Conexos", "Áreas Anotadas"],
    cols=4, figsize=(18, 6)
)
```

Figura 4.19

4.5.2 Descritores de Forma

`findContours` opera sobre a **borda** dos objetos, retornando para cada um a sequência de pontos que define seu contorno. Isso permite calcular descritores geométricos que `connectedComponentsWithStats` não fornece diretamente:

- **Perímetro** (`cv2.arcLength`)
- **Circularidade** $C = 4\pi A/P^2$ — vale 1 para círculo perfeito e é sensível a irregularidades de borda e ruídos no contorno
- **Aproximação poligonal** (`cv2.approxPolyDP`)
- **Convex hull** e **convexidade**
- **Hierarquia** — relação pai/filho entre contornos (objeto com buraco interno)

A Figure 4.20 exibe os mesmos objetos com circularidade anotada.

```python
contornos, hierarquia = cv2.findContours(
    img_final, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
)

img_contornos  = cv2.cvtColor(img_coins_gray, cv2.COLOR_GRAY2BGR)
img_circulares = img_contornos.copy()

print(f"Contornos detectados: {len(contornos)}")
print(f"\n{'ID':>4} {'Área':>8} {'Perímetro':>10} {'Circularidade':>14}")
print("-" * 42)

for i, cnt in enumerate(contornos, start=1):
    area  = cv2.contourArea(cnt)
    perim = cv2.arcLength(cnt, closed=True)
    circ  = (4 * np.pi * area / perim**2) if perim > 0 else 0
    M     = cv2.moments(cnt)
    cx    = int(M["m10"] / M["m00"]) if M["m00"] > 0 else 0
    cy    = int(M["m01"] / M["m00"]) if M["m00"] > 0 else 0

    print(f"{i:>4} {area:>8.0f} {perim:>10.1f} {circ:>14.3f}")

    cv2.drawContours(img_contornos,  [cnt], -1, (0, 255, 0), 3)
    cv2.drawContours(img_circulares, [cnt], -1, (0, 255, 0), 3)
    for cor, esp in [((0,0,0), 8), ((255,0,0), 3)]:
        cv2.putText(img_circulares, f"{circ:.2f}",
                    (cx - 80, cy + 15),
                    cv2.FONT_HERSHEY_SIMPLEX, 3.6, cor, esp, cv2.LINE_AA)

mm.show(
    [img_final, img_contornos, img_circulares],
    titles=["Segmentação Final", "Contornos", "Circularidade"],
    cols=3, figsize=(18, 6)
)
```

Figura 4.20

4.5.3 Conexão com Detecção de Objetos Moderna

Os descritores extraídos acima — ***bounding box***, **centróide**, **área** e **circularidade** — constituem a ponte entre a segmentação morfológica clássica e os modelos modernos de detecção de objetos.

Detectores baseados em aprendizado profundo, como a família **YOLO** (*You Only Look Once*) (Redmon et al., 2016), operam diretamente sobre imagens coloridas e produzem, para cada objeto detectado, uma **caixa delimitadora** (*bounding box*) com coordenadas (x, y, w, h) e uma **pontuação de confiança** — a mesma representação gerada por `connectedComponentsWithStats`. A diferença fundamental é que o YOLO aprende a detectar e classificar objetos simultaneamente, sem etapa de segmentação prévia, generalizando para categorias arbitrárias a partir de dados anotados.

A Figure 4.21 ilustra como as *bounding boxes* obtidas por morfologia podem ser exportadas no formato YOLO para treinar ou avaliar um detector:

```python
H_img, W_img = img_coins_gray.shape
img_bbox = cv2.cvtColor(img_coins_gray, cv2.COLOR_GRAY2BGR)

CLASSE = 0   # 0 = moeda (única categoria neste exemplo)

print(f"{'cls':>4} {'cx_n':>8} {'cy_n':>8} {'w_n':>8} {'h_n':>8}  ← formato YOLO")
print("-" * 54)

yolo_linhas = []
for i in range(1, n):
    x0 = stats[i, cv2.CC_STAT_LEFT]
    y0 = stats[i, cv2.CC_STAT_TOP]
    w  = stats[i, cv2.CC_STAT_WIDTH]
    h  = stats[i, cv2.CC_STAT_HEIGHT]

    cx_n = (x0 + w / 2) / W_img
    cy_n = (y0 + h / 2) / H_img
    w_n  = w / W_img
    h_n  = h / H_img

    yolo_linhas.append(f"{CLASSE} {cx_n:.4f} {cy_n:.4f} {w_n:.4f} {h_n:.4f}")
    print(f"{CLASSE:>4} {cx_n:>8.4f} {cy_n:>8.4f} {w_n:>8.4f} {h_n:>8.4f}")

    cv2.rectangle(img_bbox, (x0, y0), (x0 + w, y0 + h), (0, 255, 0), 4)
    cv2.putText(img_bbox, f"moeda", (x0 + 8, y0 + 60),
                cv2.FONT_HERSHEY_SIMPLEX, 3.8, (255, 0, 0), 5, cv2.LINE_AA)

# Exportar arquivo de anotação no formato YOLO
with open("moedas.txt", "w") as f:
    f.write("\n".join(yolo_linhas))
print("\nAnotação salva em moedas.txt")

mm.show(
    [img_coins_gray, img_final, img_bbox],
    titles=["Original", "Segmentação Final", "Bounding Boxes (formato YOLO)"],
    cols=3, figsize=(18, 6)
)
```

Figura 4.21

O formato YOLO armazena cada objeto em uma linha com cinco valores: `classe cx cy w h`, todos normalizados para $[0, 1]$ em relação às dimensões da imagem. A **classe** é um inteiro que mapeia para um nome definido em `classes.txt` (p. ex., `0 → moeda`). Quando há múltiplas categorias — por exemplo, moeda de ouro (0), moeda de prata (1) e disco plástico (2) — basta atribuir o inteiro correspondente a cada componente antes de exportar, mantendo exatamente o mesmo formato de saída.

O fluxo morfológico apresentado neste capítulo — segmentação → rotulação → extração de *bounding boxes* — corresponde exatamente à etapa de **anotação** (*labeling*) necessária para criar conjuntos de dados de treinamento para detectores como YOLO. Ferramentas como o **Label Studio** e o **Roboflow** automatizam

esse processo para imagens complexas, mas a lógica subjacente é a mesma: associar a cada objeto uma caixa delimitadora e uma classe. Em cenários controlados (fundo uniforme, objetos bem separados), a abordagem morfológica pode substituir ou inicializar a anotação manual, reduzindo significativamente o esforço de rotulação.

4.5.4 Segmentação por *K-Means*

O ***k-means*** agrupa pixels em k *clusters* minimizando a inércia (soma das distâncias quadráticas intra-cluster):

$$J = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2 \tag{4.14}$$

O algoritmo itera entre dois passos até que os centróides variem muito pouco ou seja atingido o número máximo de iterações: **(1) atribuição** — cada pixel ao centróide mais próximo; **(2) atualização** — cada centróide recalculado como média dos pixels atribuídos. A escolha de k é auxiliada pelo **método do cotovelo**: plota-se J em função de k e identifica-se o ponto de inflexão (cotovelo), a partir do qual o aumento de k traz reduções apenas marginais na inércia, evitando o sobreajuste (*overfitting*).

A Figure 4.22 mostra o método do cotovelo e a Figure 4.23 a segmentação resultante:

```
pixels   = img_coins_gray.reshape(-1, 1).astype(np.float32)
inercias = []
ks       = list(range(2, 9))
resultados = []  # guarda (labels, centers) para reusar no mm.show

for k in ks:
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.2)
    _, labels, centers = cv2.kmeans(pixels, k, None, criteria, 5, cv2.KMEANS_RANDOM_CENTERS)
    # inércia: distância de cada pixel ao centróide atribuído
    dists = (pixels - centers[labels.flatten()])**2
    inercias.append(float(dists.sum()))
    resultados.append((labels, centers))

imgs, titles = [], []
for k, inercia, (labels, centers) in zip(ks, inercias, resultados):
    seg = centers[labels.flatten()].reshape(img_coins_gray.shape).astype(np.uint8)
    imgs   += [seg]
    titles += [f"k={k}  J={inercia:.2e}"]

mm.show(imgs, titles=titles, cols=4, rows=2, figsize=(12, 12))
```

Figura 4.22

```
def kmeans_seg(img, k):
    pixels   = img.reshape(-1,1).astype(np.float32)
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.2)
    _, labels, centers = cv2.kmeans(pixels, k, None, criteria, 5, cv2.KMEANS_RANDOM_CENTERS)
    return centers[labels.flatten()].reshape(img.shape).astype(np.uint8)

imgs_km   = [img_coins_gray] + [kmeans_seg(img_coins_gray, k) for k in [2, 4, 6]]
titles_km = ["Original"] + [f"k-means k={k}" for k in [2, 4, 6]]
mm.show(imgs_km, titles=titles_km, cols=4, figsize=(18, 6))
```

Figura 4.23

4.6 Aplicação Prática: Pipeline Completo de Contagem

Reunindo as técnicas do capítulo em um *pipeline* de **contagem automática de moedas**, problema clássico em controle de qualidade industrial e visão robótica.

Avaliação: IoU (*Intersection over Union*)

Para avaliar a qualidade da segmentação com máscara de referência (*ground truth*):

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} \tag{4.15}$$

onde A é a segmentação obtida e B é a referência. IoU $= 1$ indica segmentação perfeita. Para múltiplos objetos. Em aplicações de detecção de objetos, frequentemente utiliza-se IoU ≥ 0.5 como critério mínimo de correspondência..

4.7 Resumo

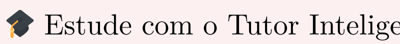
Neste capítulo foram apresentadas as técnicas de segmentação e morfologia matemática, fechando o ciclo do PDI no domínio espacial:

- **Otsu e pré-processamento:** O algoritmo CLAHE provou-se superior na indução de separação bimodal no histograma ($\sigma_B^2 \approx 2{,}47 \times 10^3$, $T^* = 122$), simplificando a binarização de imagens com iluminação não uniforme.
- **Erosão e dilatação:** Operadores morfológicos fundamentais baseados na busca por mínimo/máximo local em uma vizinhança definida pelo elemento estruturante B. São duais pelo complemento e implementados no pacote prático como `mm.ero` e `mm.dil`.
- **Abertura e fechamento:** Composições estritas (erosão $\rightarrow$ dilatação e dilatação $\rightarrow$ erosão) com propriedades de idempotência, fundamentais para suprimir ruídos externos e preencher lacunas internas, respectivamente.
- **Reconstrução morfológica:** Operação geodésica iterativa que propaga um marcador estritamente dentro dos limites de uma máscara. Constitui o motor por trás de filtros que não deformam a geometria dos objetos, como `mm.clohole` e `mm.edgeoff`.
- **Pipeline de limpeza binária:** Fluxo consolidado como: CLAHE $\rightarrow$ Otsu $\rightarrow$ abertura (filtra ruído de fundo) $\rightarrow$ `mm.clohole` (preenche cavidades) $\rightarrow$ abertura restrita (suaviza artefatos espúrios) $\rightarrow$ `mm.edgeoff` (filtra objetos tocando a borda).
- **Morfologia em tons de cinza:** Extensão algébrica com mínimo/máximo ponderado, viabilizando o gradiente morfológico e filtros de *top-hat* para extração de topologia.
- **Componentes conexos e descritores:** Rotulagem espacial (`cv2.connectedComponentsWithStats`) acoplada à extração de fronteiras ativas, gerando métricas como área, circularidade e centróides.
- ***Watershed* e *k-means*:** Algoritmos complementares; o primeiro focado na separação topográfica de objetos espacialmente sobrepostos, e o segundo no agrupamento estatístico por similaridade de intensidades.

O Capítulo 5 abordará o **domínio da frequência** (Transformada de Fourier, filtragem espectral) e os fundamentos de **compressão de imagens** (DCT, JPEG, *wavelets*).

4.8 🤖 Uso do NotebookLM como Tutor Complementar

Nesta edição, incentiva-se o uso do **NotebookLM** como ferramenta complementar de aprendizagem. Baseado em inteligência artificial, o sistema utiliza exclusivamente os documentos fornecidos pelo autor como fonte de conhecimento, produzindo respostas alinhadas ao conteúdo e à abordagem adotada ao longo do livro.



🚀 ACESSAR NOTEBOOKLM: CAPÍTULO 04

4.9 Lista de Exercícios

1. **(10%)** Implemente manualmente o critério de Otsu sem usar `cv2.threshold`: calcule $\sigma_B^2(T)$ para todos os $T \in [0, 255]$ usando `mm.hist`, identifique T^* e compare com o valor do OpenCV. Plote σ_B^2 em função de T e marque o máximo.
2. **(15%)** Aplique limiarização adaptativa com blocos de tamanho 11, 31 e 51 à imagem Lena com gradiente de iluminação simulado (`np.linspace` adicionado à imagem). Compare visualmente com Otsu global e explique por que a adaptativa é superior nesse cenário.
3. **(15%)** Execute o *pipeline watershed* na imagem de moedas variando o limiar da transformada de distância (0.3, 0.5, 0.7× máximo). Explique como esse parâmetro afeta a separação de objetos sobrepostos e a criação de sobre-segmentação.
4. **(15%)** Usando `mm.drawImage` e `mm.drawImageKernel`, crie uma demonstração visual passo a passo da erosão de uma imagem binária 7×7 com elemento estruturante 3×3 quadrado — mostrando para cada pixel central se o elemento estruturante está completamente contido em A ou não.
5. **(15%)** Demonstre a **dualidade erosão–dilatação** experimentalmente com `mm.ero`, `mm.dil` e `mm.bnot`: verifique que $(A \ominus B)^c = A^c \oplus \hat{B}$. Calcule a diferença pixel a pixel entre os dois lados e exiba com `mm.histImg`.
6. **(15%)** Implemente o **gradiente morfológico** manualmente usando `mm.ero` e `mm.dil` e compare com `cv2.morphologyEx(img, cv2.MORPH_GRADIENT, B)`. Aplique com elementos estruturantes de forma e tamanho diferentes (quadrado 3×3, disco 5×5, linha 1×9) e explique as diferenças na resposta de bordas.
7. **(15%)** Construa um *pipeline* completo para contar e classificar as moedas por tamanho (pequena, média, grande) usando área e circularidade como critérios. Calcule o **IoU** entre sua segmentação e uma máscara de referência criada manualmente. Apresente os resultados em tabela e com `mm.show` em grade.

Referências do Capítulo

A fundamentação teórica deste capítulo baseia-se nas seguintes obras:

- Gonzalez; Woods (2018) para os conceitos de segmentação, limiarização de Otsu, morfologia matemática e descritores de forma.
- Matheron (1975) e Serra (1982) para a fundamentação teórica original, formulação algébrica e desenvolvimento da Morfologia Matemática.
- Szeliski (2022) para *watershed*, *k-means* aplicado a imagens e avaliação quantitativa com a métrica IoU.
- Bradski; Kaehler (2008) para a implementação prática com a biblioteca OpenCV (`cv2.watershed`, `cv2.connectedComponentsWithStats`) e o pacote didático `morph.py`.
- Redmon et al. (2016) para a introdução ao formato de anotação YOLO e sua conexão com os descritores de *bounding box* extraídos por morfologia.

4.10 💻 Parte Prática com Exercícios de Programação

🎯 Objetivo deste Caderno

O caderno permite desenvolver, validar, organizar e testar soluções de **Exercícios de Programação (EPs)** em ambientes interativos, como o Colab, com os mesmos casos de teste do Moodle, copiando para lá apenas na hora de registrar a nota oficial.

Download

Baixe `morph.py` e `testsuite.py` executando a célula abaixo:

```
import os, sys, importlib, inspect, urllib.request

# URLs do repositório
BASE_URL = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph"
for f in ["morph.py", "testsuite.py"]:
    if not os.path.exists(f):
        urllib.request.urlretrieve(f"{BASE_URL}/{f}", f)

import morph, testsuite
importlib.reload(morph); importlib.reload(testsuite)
from morph import mm
from testsuite import TestSuite

print(f" Ambiente pronto. Morph: {morph.__version__} | TestSuite: {testsuite.__version__}")
```

```
 Ambiente pronto. Morph: 1.1.0 | TestSuite: 1.1.0
```

Executando os Testes

Para rodar os testes, execute `TestSuite("EP01_01.extensão").run()` numa nova célula, trocando a extensão pela da linguagem usada (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). O sistema baixa os casos de teste do GitHub, executa o programa e calcula a nota automaticamente.

Exemplo de teste de `sqrt` em Python, com `timeit` isolando cada operação:

```
%%writefile EP02_01.py
# Código Python
x1,y1,x2,y2 = int(input()), int(input()), int(input()), int(input())
# Cálculo das diferenças
dx = abs(x2 - x1)
dy = abs(y2 - y1)

# 1. Distância Euclidiana (L2)
dist_euclidiana = (dx**2 + dy**2)**0.5

# 2. Distância City-block / Manhattan (L1)
dist_city_block = dx + dy

# 3. Distância Chessboard / Chebyshev (Linf)
dist_chessboard = max(dx, dy)

# Saída formatada conforme os casos de teste
print(f"{dist_euclidiana:.2f}")
print(f"{dist_city_block:.2f}")
print(f"{dist_chessboard:.2f}")
```

```
Writing EP02_01.py
```

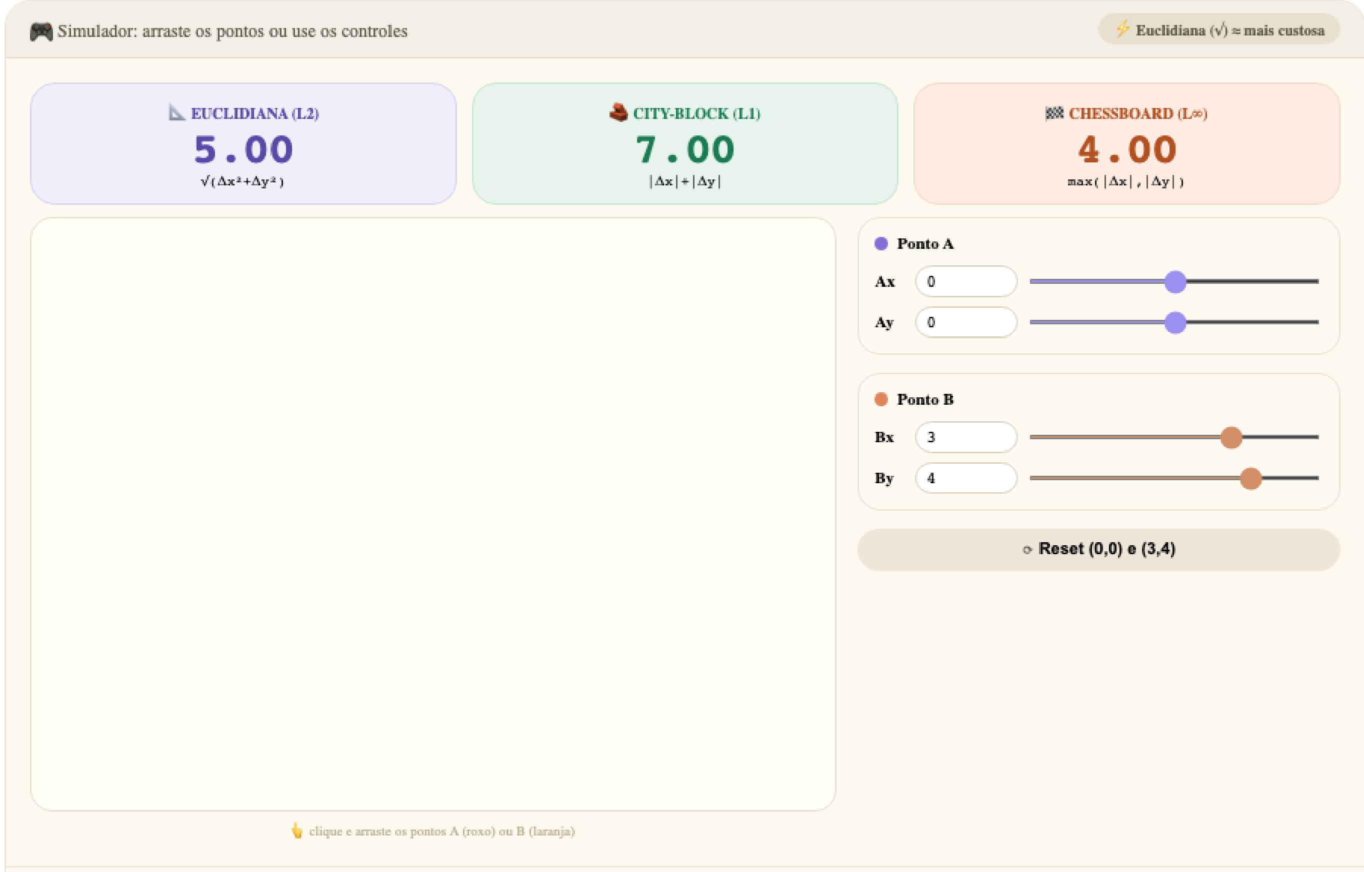


Figura 4.24: Simulador: Distâncias Euclidiana, *City-block* e *Chessboard*

```
import numpy as np
np.set_printoptions(linewidth=100, edgeitems=3, threshold=1000)
```

```
print("Substitui emoji Unicode literal  por comando")
```

```
Substitui emoji Unicode literal  por comando
```

```
import numpy as np
print(mm.drawImage(np.zeros((5, 5))))
```

```
  0   0   0   0   0
  0   0   0   0   0
  0   0   0   0   0
  0   0   0   0   0
  0   0   0   0   0
```

```
TestSuite("EP02_01.py").run()
```

Capítulo 5

Transformadas e Compressão

> *"A Fourier transform does not change the information content of a signal; it changes the representation."* — R. C. Gonzalez & R. E. Woods

Nos capítulos anteriores, todas as operações foram realizadas **no domínio espacial**: filtros de convolução, morfologia e segmentação atuam diretamente sobre os valores de intensidade dos pixels. Este capítulo apresenta uma perspectiva complementar e poderosa — o **domínio da frequência** — que revela como a imagem é construída a partir de padrões oscilatórios de diferentes escalas.

A ideia central é simples: **qualquer imagem pode ser decomposta em uma soma de senoides bidimensionais**, cada uma com frequência, amplitude e fase próprias. Baixas frequências correspondem a variações suaves (fundo, iluminação global); altas frequências correspondem a bordas, texturas e ruído.

Esta decomposição abre três caminhos práticos:

1. **Filtragem espectral** — suprimir ou realçar seletivamente faixas de frequência;
2. **Análise multi-escala com *wavelets*** — capturar estruturas em diferentes resoluções e posições;
3. **Compressão** — eliminar coeficientes imperceptíveis ao olho humano, reduzindo o volume de dados.

5.1 Objetivos

Ao concluir este capítulo, você será capaz de:

- **Interpretar o espectro de Fourier** de uma imagem: distinguir magnitude de fase, localizar baixas e altas frequências, e ler o espectro de forma intuitiva;
- **Aplicar o Teorema da Convolução** para implementar filtragem eficiente via FFT, compreendendo o ganho computacional em relação à convolução direta;
- **Projetar e comparar filtros espectrais** — passa-baixa (Ideal, Gaussiano, Butterworth), passa-alta e notch — com foco nos seus efeitos visuais e nos artefatos que produzem;
- **Compreender wavelets e multirresolução**: interpretar as subbandas LL/LH/HL/HH e relacionar a análise *wavelet* às camadas de CNNs;
- **Entender o pipeline JPEG**: desde a DCT em blocos 8×8 até a quantização perceptual e os artefatos de compressão;
- **Escolher o formato de imagem** adequado (JPEG, PNG, WebP) com base no conteúdo e na aplicação.

5.2 Configuração do Ambiente

```
import os, importlib, urllib.request
import numpy as np
import matplotlib.pyplot as plt
import cv2
from scipy import ndimage
```

```
BASE_URL = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph"
for f in ["morph.py"]:
    if not os.path.exists(f):
        urllib.request.urlretrieve(f"{BASE_URL}/{f}", f)

import morph
importlib.reload(morph)
from morph import mm

print("  Ambiente pronto")
```

```
Ambiente pronto
```

5.3 Transformada de Fourier Discreta 2D

A análise de Fourier fundamenta-se em um princípio poderoso: qualquer sinal periódico admite representação como soma de senoides com diferentes frequências, amplitudes e fases. Ver exemplo em uma dimensão na Figure 5.1.

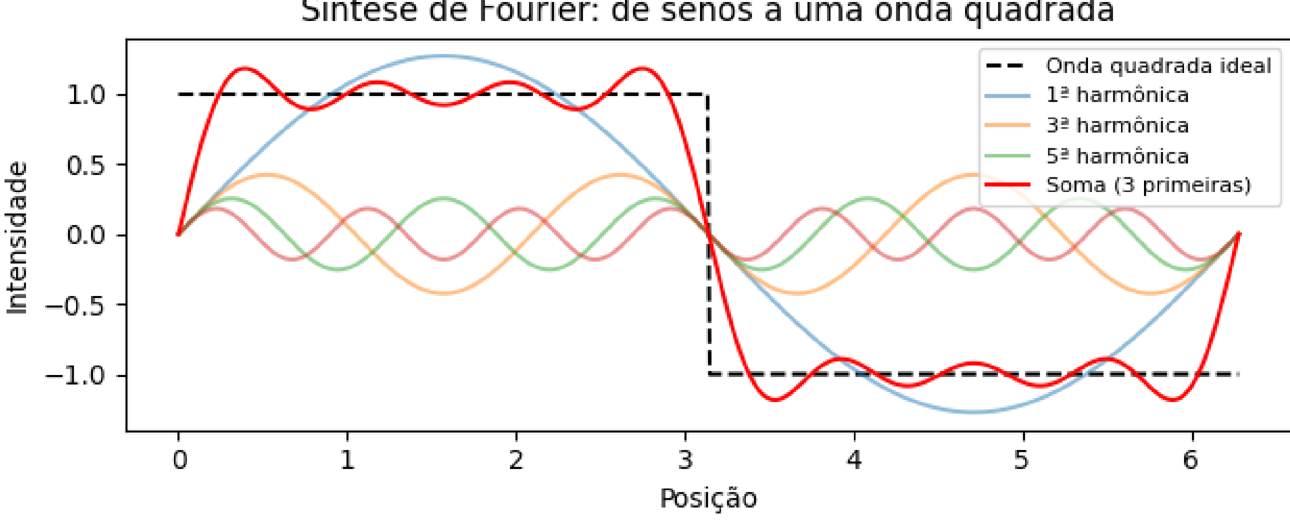


Figura 5.1: Decomposição de Fourier 1D: uma onda quadrada (linha tracejada) é aproximada pela soma das primeiras senoides (linhas coloridas). Quanto mais termos, melhor a aproximação.

Aplicada a imagens digitais, essa ideia transforma uma matriz de valores de intensidade $f(x, y)$ em uma representação no **domínio da frequência**, revelando quais padrões oscilatórios compõem a cena visual.

5.3.1 Fenômeno: o que uma imagem "parece" no domínio da frequência?

Observe o que acontece quando se transforma uma fotografia: ela se transforma em um padrão radial brilhante ao centro, com energia decaindo progressivamente em direção às bordas. Esse padrão é o **espectro de magnitude** da imagem — uma radiografia da distribuição de frequências.

Região do espectro	O que representa	Exemplos visuais
Centro (baixas freq.)	Variações lentas — iluminação, cor de fundo, forma global	Gradientes suaves, blocos uniformes
Meio (médias freq.)	Texturas, padrões repetitivos	Tecidos, tijolos, grama
Bordas (altas freq.)	Variações abruptas — contornos, ruído	Bordas de objetos, grão de imagem

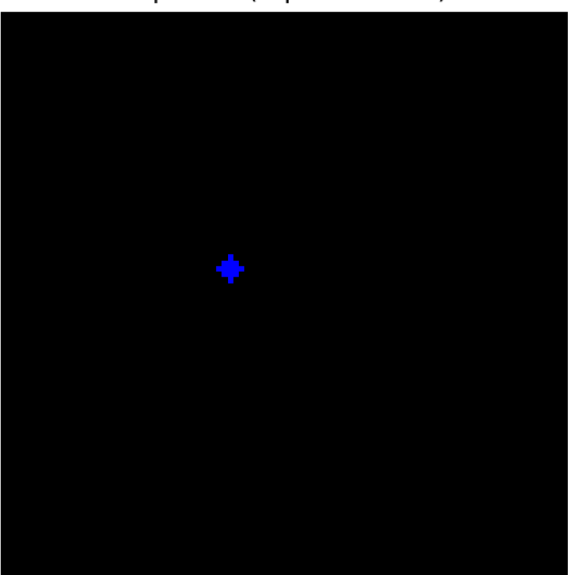
Intuição-chave: suprimir o centro do espectro remove a estrutura global e enfatiza bordas; suprimir as bordas suaviza a imagem e remove ruído. É precisamente isso que os filtros espectrais fazem — mas com precisão cirúrgica impossível no domínio espacial.

5.3.2 O Experimento da Grade: Construindo a Imagem do Zero

Antes de mergulhar nas fórmulas complexas, vamos fazer o caminho inverso (IDFT). O que acontece se tivermos um espectro completamente vazio (preto) e acendermos **apenas um único ponto** brilhante (um coeficiente de frequência)?

O resultado no espaço real é uma grade perfeitamente senoidal. A posição desse ponto no espectro define a **direção** e a **frequência** (espessura) dessa grade.

```
N_grid = 100
espectro_vazio = np.zeros((N_grid, N_grid), dtype=complex)

# Acendendo um único ponto (frequência) fora do centro
u0, v0 = 10, 5
espectro_vazio[N_grid//2 - v0, N_grid//2 - u0] = 1000

# Retornando para o domínio espacial (IDFT)
onda_2d = np.real(np.fft.ifft2(np.fft.ifftshift(espectro_vazio)))

onda_vis = cv2.normalize(onda_2d, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)
espectro_vis = cv2.normalize(np.abs(espectro_vazio), None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

# Destaque visual do ponto
espectro_color = cv2.cvtColor(espectro_vis, cv2.COLOR_GRAY2BGR)
cv2.circle(espectro_color, (N_grid//2 - u0, N_grid//2 - v0), 2, (0, 0, 255), -1)

mm.show([espectro_color, onda_vis], titles=["Espectro (1 ponto ativo)", "Onda 2D Resultante (IDFT)"], co
```



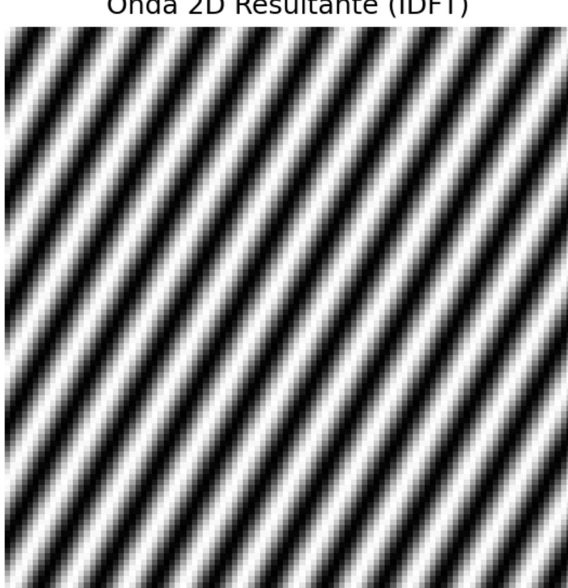


Figura 5.2: Toda frequência no espectro (ponto isolado) corresponde a uma onda senoidal 2D rotacionada no domínio espacial.

5.3.3 Definição Matemática

Dada uma imagem $f(x,y)$ de dimensões $M \times N$, sua **Transformada de Fourier Discreta 2D** (DFT) é definida como:

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y)\, e^{-j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)} \tag{5.1}$$

onde $u = 0,1,\dots,M-1$ e $v = 0,1,\dots,N-1$ são as **frequências discretas** nas direções horizontal e vertical, respectivamente. O exponencial complexo $e^{-j2\pi(ux/M+vy/N)}$ representa uma oscilação bidimensional de frequência $(u/M, v/N)$ ciclos por pixel.

A **Transformada Inversa de Fourier Discreta 2D** (IDFT) recupera a imagem original a partir de seu espectro:

$$f(x,y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v)\, e^{j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)} \tag{5.2}$$

5.3.4 As Funções de Base da DCT

A **frequência espacial** $(u/M, v/N)$ representa a taxa de variação espacial da intensidade ao longo da imagem.

Valores baixos correspondem a variações suaves e estruturas globais; valores altos correspondem a mudanças rápidas de intensidade, como bordas, texturas e detalhes finos.

Em termos discretos, os índices (u,v) indicam quantos ciclos da oscilação ocorrem ao longo da largura e da altura da imagem.

Cada bloco 8×8 da imagem é decomposto em uma combinação ponderada de **64 padrões de base** da DCT — funções cossenoidais de frequência crescente em x e em y. O coeficiente $C(u,v)$ representa o "peso" do padrão de frequência (u,v) naquele bloco específico.

A figura abaixo exibe as 64 bases: o canto superior esquerdo corresponde ao componente DC (padrão uniforme); ao avançar para a direita e para baixo, a frequência espacial aumenta progressivamente.

> **i** O que é o componente DC?
>
> O coeficiente $F(0,0)$ — chamado **DC** (*Direct Current*) — corresponde à **média global** de todos os pixels. Na grande maioria das imagens naturais, corresponde ao valor de maior magnitude no espectro — imagens com variações abruptas e periódicas podem concentrar energia em outras frequências. Os demais coeficientes $(u,v \neq 0,0)$ representam variações em torno da média.
> Após **fftshift** — deslocamento circular que move o DC para o centro —, o espectro fica intuitivo: centro = baixas frequências, bordas = altas frequências.

5.3.5 Implementação e Visualização

5.3.6 O que a Magnitude e a Fase carregam?

Uma das demonstrações mais reveladoras em PDI é trocar os espectros de magnitude e fase entre duas imagens distintas e reconstruir cada uma. O resultado mostra que:

- **A fase carrega a estrutura semântica** — contornos, posição dos objetos, geometria da cena;
- **A magnitude carrega o conteúdo energético e textural** — contraste, iluminação geral.

Quando reconstruímos uma imagem com a **magnitude de A** mas a **fase de B**, o resultado se parece com B — mesmo usando a "energia" de A. A fase possui papel predominante na preservação da estrutura geométrica

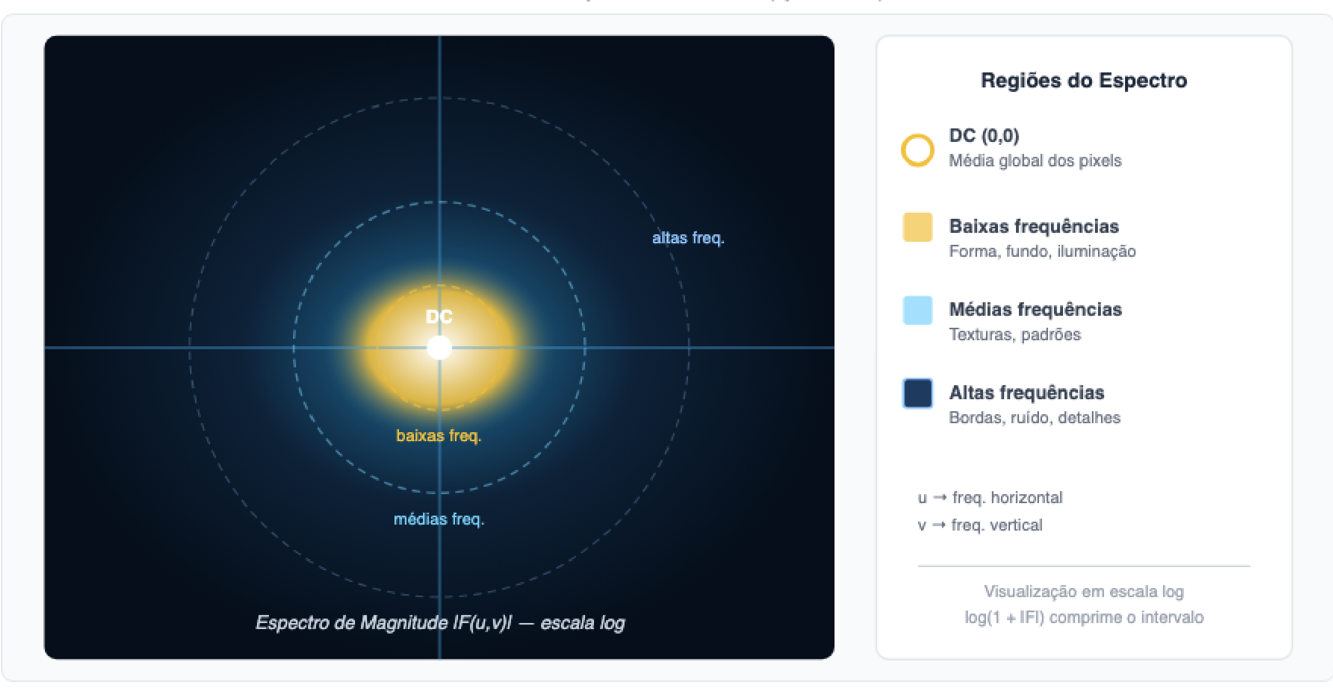


Figura 5.3: Diagrama conceitual do espectro de Fourier 2D centrado.

e organização espacial da imagem, enquanto a magnitude controla a distribuição energética, contraste e textura. Ver um exemplo na Figure 5.4.

> **Analogia auditiva:** no áudio, a fase determina a percepção de posição (esquerda/direita). Na imagem, ela determina a posição e forma dos objetos. Inverter a fase de uma música torna-a irreconhecível mesmo que a magnitude (timbre) permaneça intacta.

```
#   Experimento: A Importância da Fase
#   Carregamento da imagem
url     = "https://upload.wikimedia.org/wikipedia/commons/2/25/GAZI.MD.AHAD_11.jpg"
caminho = "imagens/coins.jpg"

if not os.path.exists(caminho):
    os.makedirs("imagens", exist_ok=True)
    img_obj = mm.read(url, pil=True)
    mm.write(img_obj, caminho)
else:
    img_obj = mm.read(caminho, pil=True)

img_color = np.array(img_obj)
img_gray  = mm.gray(img_color)

img_a = cv2.resize(img_gray, (400, 400))

# Criar uma imagem B sintética (padrão geométrico)
img_b = np.zeros((400, 400), dtype=np.uint8)
cv2.rectangle(img_b, (100, 100), (300, 300), 255, -1)
cv2.circle(img_b, (200, 200), 150, 128, 10)
```

```
FA = np.fft.fft2(img_a)
FB = np.fft.fft2(img_b)

# Troca de Fase
rec_A_mag_B_fase = np.real(np.fft.ifft2(np.abs(FA) * np.exp(1j * np.angle(FB))))
rec_B_mag_A_fase = np.real(np.fft.ifft2(np.abs(FB) * np.exp(1j * np.angle(FA))))

mm.show(
    [img_a, img_b, rec_A_mag_B_fase, rec_B_mag_A_fase],
    titles=["Imagem A", "Imagem B", "Mag(A) + Fase(B)", "Mag(B) + Fase(A)"],
    cols=4, figsize=(16, 4)
)

print(" Interpretação: Observe que a estrutura (formas) segue a FASE, não a magnitude.")
```

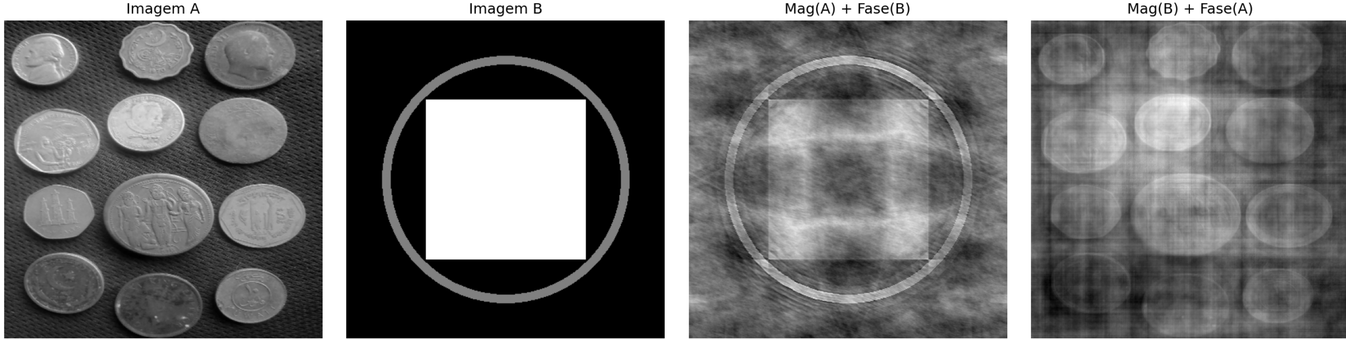


Figura 5.4: Experimento de troca de fase: Imagem A (moedas) e Imagem B (padrão geométrico) reconstruídas com magnitudes e fases trocadas. O resultado confirma que a estrutura visual segue a fase — a imagem reconstruída com a fase de B se parece com B, independentemente de qual magnitude foi usada.

```
Interpretação: Observe que a estrutura (formas) segue a FASE, não a magnitude.
```

5.3.7 Simulador: Reconstruindo Sinais com Senoides

Antes de prosseguir para imagens 2D, o simulador interativo da Figure 5.5 permite explorar a essência da análise de Fourier 1D: **qualquer forma de onda pode ser reconstruída somando-se senoides simples**.

Observe como, ao adicionar mais termos, a soma (curva preta) converge progressivamente para a forma alvo (tracejada). O espectro abaixo mostra as amplitudes de cada frequência — é a "receita" da forma de onda.

> **Explore ativamente:** *(i)* Com quantos termos a onda quadrada fica visualmente aceitável? *(ii)* Qual das três formas converge mais rapidamente — por quê? *(iii)* O que acontece com o espectro ao trocar de onda quadrada para triangular?

5.4 Teorema da Convolução

O **Teorema da Convolução** estabelece uma das relações mais importantes entre os domínios espacial e de frequência:

$$f(x,y) * h(x,y) \ \mathcal{F} \ F(u,v) \cdot H(u,v) \tag{5.3}$$

Isso significa que, em vez de **deslizar um *kernel* sobre cada pixel da imagem = convolução** (o que é lento para *kernels* grandes), podemos:

1. Transformar a imagem para o domínio da frequência via FFT;

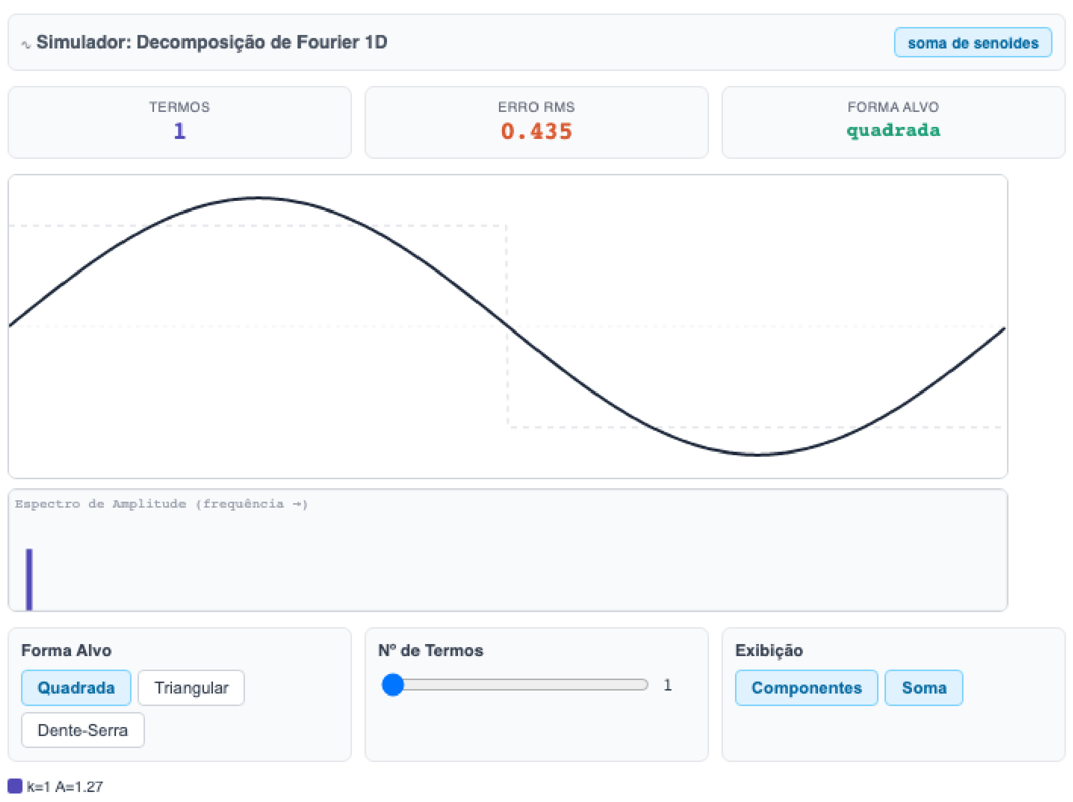


Figura 5.5: Simulador interativo da decomposição de Fourier 1D: visualização da soma de senoides com diferentes frequências, amplitudes e fases. Adicione termos e observe a convergência para formas de onda arbitrárias.

2. **Multiplicar** ponto a ponto pelo espectro do filtro;
3. Transformar de volta via IFFT.

5.4.1 Eficiência Computacional: DFT vs Convolução Direta

A implicação prática imediata é de natureza algorítmica. Para um filtro de tamanho $K \times K$ aplicado a uma imagem $N \times N$:

Tabela 5.2: Comparação de complexidade computacional entre convolução direta e filtragem via FFT.

Método	Complexidade	Exemplo ($N = 512$, $K = 51$)
Convolução direta	$O(N^2K^2)$	$\approx 682 \times 10^6$ ops
Filtragem via FFT	$O(N^2 \log N)$	$\approx 2{,}36 \times 10^6$ ops

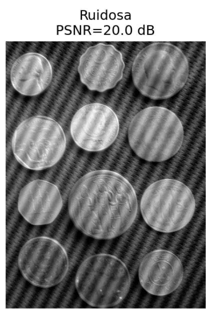
A filtragem via FFT torna-se significativamente mais eficiente quando o filtro h é grande. O fluxo computacional é:

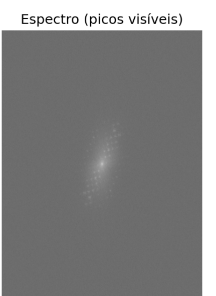
$$g = \mathcal{F}^{-1}[\mathcal{F}(f) \cdot \mathcal{F}(h)] \tag{5.4}$$

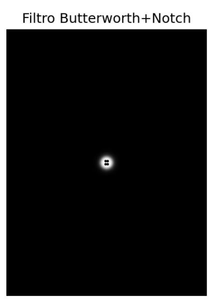
```python
np.random.seed(42)
h_img, w_img = img_gray.shape
X2, Y2 = np.meshgrid(np.arange(w_img), np.arange(h_img))

#   1. Ruído misto
ruido_gauss  = np.random.normal(0, 15, img_gray.shape)
ruido_period = 30 * np.sin(2 * np.pi * (15 * X2/w_img + 10 * Y2/h_img))
img_noisy    = np.clip(img_gray.astype(float) + ruido_gauss + ruido_period, 0, 255).astype(np.uint8)

#   2. Espectro
F_n    = np.fft.fftshift(np.fft.fft2(img_noisy.astype(np.float64)))
mag_n  = cv2.normalize(np.log1p(np.abs(F_n)), None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

#   3. Filtro Butterworth (remove ruído gaussiano)
D_n    = distancia_centro(h_img, w_img)
H_bw   = 1.0 / (1.0 + (D_n / 60.0) ** 6)    # D0=60, n=3

#   4. Máscara notch (remove picos periódicos)
mascara_dn = np.ones((h_img, w_img), dtype=np.float64)
for dy, dx in [(+15,+10),(-15,-10),(+15,-10),(-15,+10)]:
    mascara_dn = suprimir_pico(mascara_dn, h_img//2+dy, w_img//2+dx, r=12)

H_total = H_bw * mascara_dn

#   5. Filtragem e reconstrução
img_den = np.real(np.fft.ifft2(np.fft.ifftshift(F_n * H_total)))
img_den = np.clip(img_den, 0, 255).astype(np.uint8)

#   6. Métricas
psnr_n, ssim_n = cv2.PSNR(img_gray, img_noisy), ssim(img_gray, img_noisy)
psnr_d, ssim_d = cv2.PSNR(img_gray, img_den),   ssim(img_gray, img_den)

print(f"{'Métrica':>10} | {'Imagem ruidosa':>15} | {'Após filtragem':>15}")
print("-"*46)
print(f"{'PSNR (dB)':>10} | {psnr_n:>15.2f} | {psnr_d:>15.2f}")
print(f"{'SSIM':>10} | {ssim_n:>15.4f} | {ssim_d:>15.4f}")
```

```
H_vis_dn = cv2.normalize((H_total*255).astype(np.uint8), None, 0, 255, cv2.NORM_MINMAX)

mm.show(
    [img_gray, img_noisy, mag_n, H_vis_dn, img_den],
    titles=["Original",
            f"Ruidosa\nPSNR={psnr_n:.1f} dB",
            "Espectro (picos visíveis)",
            "Filtro Butterworth+Notch",
            f"Restaurada\nPSNR={psnr_d:.1f} dB  SSIM={ssim_d:.3f}"],
    cols=5, figsize=(18, 4)
)
```

```
   Métrica |  Imagem ruidosa |  Após filtragem
----------------------------------------------
 PSNR (dB) |           19.97 |           21.21
      SSIM |          0.3478 |          0.6892
```







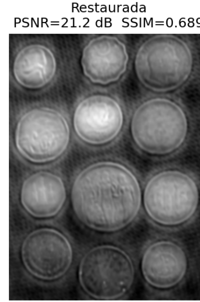


Figura 5.6: Pipeline completo: Butterworth + Notch para remoção de ruído misto.

! O Perigo da Convolução Circular (*Wrap-around Error*)

A DFT pressupõe que a imagem se repete infinitamente como um azulejo. Se aplicarmos um filtro de frequência sem esticar a imagem com bordas de zeros (*zero-padding*), o topo da imagem se mistura com a base, e a esquerda com a direita. O *padding* evita esse vazamento.

```
# Simulação de um filtro de deslocamento brutal
H_shift = np.zeros_like(img_gray, dtype=complex)
for u in range(M):
    for v in range(N):
        H_shift[u, v] = np.exp(-1j * 2 * np.pi * (u*80/M + v*80/N)) # Desloca 80 pixels

# Filtragem SEM padding (causa o wrap-around)
F_img = np.fft.fft2(img_gray)
img_vazada = np.real(np.fft.ifft2(F_img * H_shift))

img_vazada_vis = cv2.normalize(img_vazada, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)
mm.show([img_gray, img_vazada_vis], titles=["Original", "Filtragem s/ Padding (Vazamento)"], cols=2, fig
```

Figura 5.7: Sem padding, um deslocamento severo faz a imagem vazar para o lado oposto (convolução circular).

```
#   Kernel Gaussiano 11×11
sigma   = 3.0
K       = 11
ks      = np.arange(K) - K // 2
gauss1d = np.exp(-ks**2 / (2 * sigma**2))
gauss1d /= gauss1d.sum()
kernel  = np.outer(gauss1d, gauss1d)    # kernel 2D separável

#   Método 1: Convolução espacial direta
f_float  = img_gray.astype(np.float64)
conv_esp = cv2.filter2D(f_float, -1, kernel, borderType=cv2.BORDER_CONSTANT)

#   Método 2: Multiplicação em frequência (via FFT)
M, N     = f_float.shape
# Padding para convolução linear (evita aliasing circular)
Mpad     = 2 ** int(np.ceil(np.log2(M + K - 1)))
Npad     = 2 ** int(np.ceil(np.log2(N + K - 1)))

# Posiciona o kernel com a origem no (0,0) e padding com zeros
kernel_pad          = np.zeros((Mpad, Npad))
kh, kw              = kernel.shape
kernel_pad[:kh, :kw] = kernel

F_img   = np.fft.fft2(f_float,  (Mpad, Npad))
F_kern  = np.fft.fft2(kernel_pad)
conv_freq = np.real(np.fft.ifft2(F_img * F_kern))

# Recorte para compensar o deslocamento introduído pelo posicionamento do kernel
offset    = K // 2
conv_freq_crop = conv_freq[offset:offset+M, offset:offset+N]

#   Verificação numérica
```

```
diff = np.abs(conv_esp - conv_freq_crop)
print(f"Diferença máxima  (|conv_esp - conv_freq|): {diff.max():.2e}")
print(f"Diferença média   (|conv_esp - conv_freq|): {diff.mean():.2e}")
print(f"→ Teorema da Convolução verificado numericamente.")

conv_esp_vis  = cv2.normalize(conv_esp,       None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)
conv_freq_vis = cv2.normalize(conv_freq_crop, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)
diff_vis      = cv2.normalize(diff,           None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

mm.show(
    [img_gray, conv_esp_vis, conv_freq_vis, diff_vis],
    titles=[
        "Original",
        "Convolução espacial",
        "Multiplicação em frequência",
        f"Diferença (máx={diff.max():.1e})"
    ],
    cols=4, figsize=(16, 5)
)
```

```
Diferença máxima  (|conv_esp - conv_freq|): 2.56e-13
Diferença média   (|conv_esp - conv_freq|): 2.79e-14
→ Teorema da Convolução verificado numericamente.
```

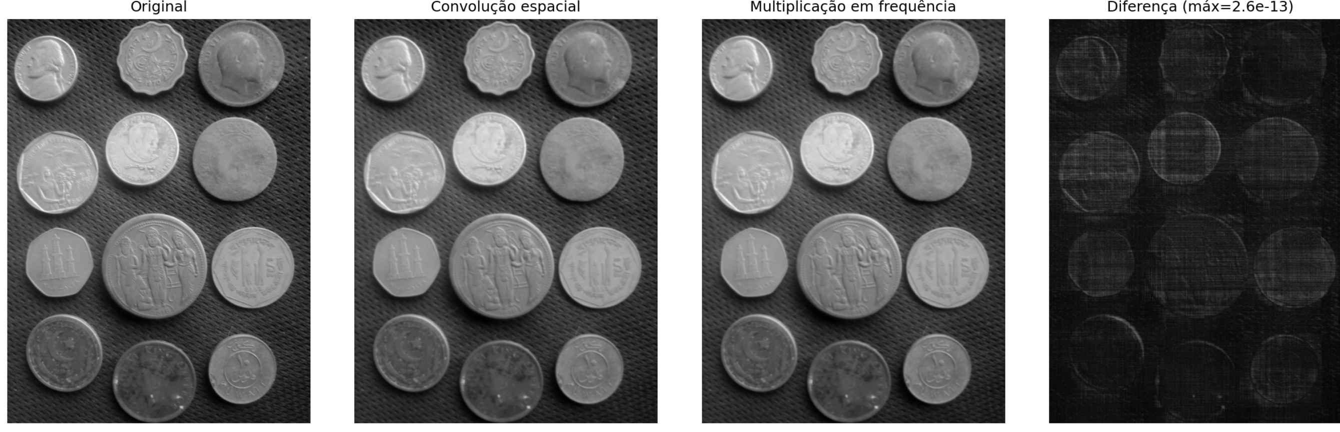


Figura 5.8: Verificação do Teorema da Convolução: a diferença pixel a pixel entre a convolução espacial (cv2.filter2D) e a multiplicação em frequência (FFT) é numericamente nula — confirmando a equivalência teórica.

5.5 Filtros no Domínio da Frequência

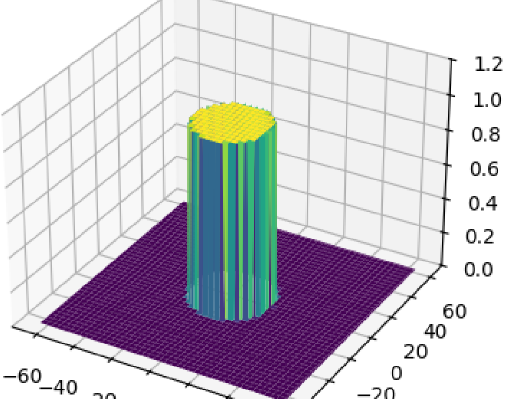
Um filtro no domínio da frequência é, essencialmente, uma **máscara aplicada ao espectro**: valores próximos de 1 deixam a frequência passar; valores próximos de 0 a atenuam. A forma dessa máscara determina o comportamento visual do filtro.

O dilema do corte abrupto: um filtro ideal (corte perfeito em D_0) parece ótimo na teoria, mas causa o **fenômeno de *Ringing*** — oscilações próximas às bordas dos objetos. Isso ocorre porque um corte abrupto no espectro corresponde, no domínio espacial, a uma convolução com uma função *sinc* de suporte infinito — que produz os anéis visíveis, ver Figure 5.9.

A solução: filtros com transição suave — Gaussiano ou Butterworth — eliminam o *ringing* ao custo de uma fronteira de corte menos precisa.

```
# Simulando o Filtro Ideal na Frequência (Cilindro) e sua representação Espacial (Sinc)
N_grid = 128
u = np.arange(-N_grid//2, N_grid//2)
U, V = np.meshgrid(u, u)
D = np.sqrt(U**2 + V**2)

# Frequência: Cilindro Ideal (1 no centro, 0 fora do raio 20)
H_freq = np.zeros((N_grid, N_grid))
H_freq[D <= 20] = 1

# Espaço: A inversa resulta na famigerada Sinc 2D
h_space = np.fft.fftshift(np.real(np.fft.ifft2(np.fft.ifftshift(H_freq))))

fig, ax = plt.subplots(1, 2, subplot_kw={'projection': '3d'}, figsize=(12, 4))
ax[0].plot_surface(U, V, H_freq, cmap='viridis', edgecolor='none')
ax[0].set_title("Frequência: Filtro Ideal (Cilindro)")
ax[0].set_zlim(0, 1.2)

ax[1].plot_surface(U, V, h_space, cmap='plasma', edgecolor='none')
ax[1].set_title("Espaço Real: Ondulações da Sinc (Causa do Ringing)")
plt.tight_layout(); plt.show()
```

Frequência: Filtro Ideal (Cilindro)



Espaço Real: Ondulações da Sinc (Causa do Ringing)

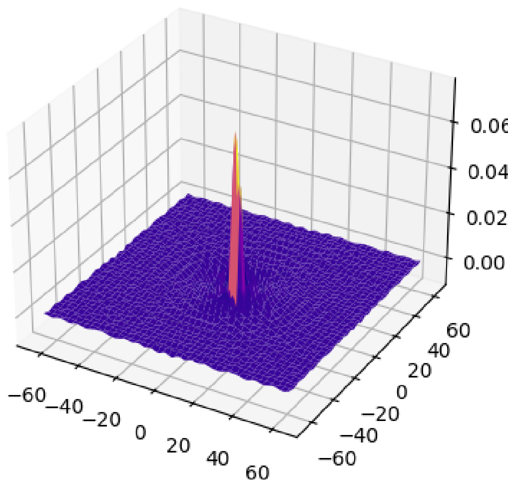


Figura 5.9: A Dualidade Perigosa: O corte abrupto na Frequência (Cilindro) transforma-se obrigatoriamente numa Sinc espacial. Suas ondulações causam o *ringing* fantasma nas bordas da imagem.

5.5.1 Filtros Passa-Baixa

Filtros passa-baixa atenuam altas frequências, suavizando a imagem e reduzindo ruído. Definem-se em termos da distância ao centro do espectro:

$$D(u,v) = \sqrt{\left(u - \frac{M}{2}\right)^2 + \left(v - \frac{N}{2}\right)^2} \tag{5.5}$$

Filtro Ideal (LPFI):

$$H_{\text{ideal}}(u,v) = \begin{cases} 1, & D(u,v) \le D_0 \\ 0, & D(u,v) > D_0 \end{cases} \tag{5.6}$$

O corte abrupto em D_0 causa o **fenômeno de Ringing**: oscilações artificiais próximas às bordas dos objetos, análogas ao comportamento da série de Fourier truncada.

Filtro Gaussiano (LPFG):

$$H_{\text{gauss}}(u, v) = e^{-D^2(u,v)/(2\sigma^2)} \tag{5.7}$$

No domínio contínuo, a transformada de Fourier de uma Gaussiana é também Gaussiana — propriedade que garante transição suave sem *ringing*.

Filtro Butterworth (LPFB) de ordem n:

$$H_{\text{BW}}(u, v) = \frac{1}{1 + [D(u, v)/D_0]^{2n}} \tag{5.8}$$

O parâmetro n controla a inclinação da transição: valores baixos ($n = 1, 2$) produzem transições suaves (sem *ringing*); valores altos ($n \geq 5$) aproximam o comportamento do filtro ideal. Ver exemplos na Figure 5.10.

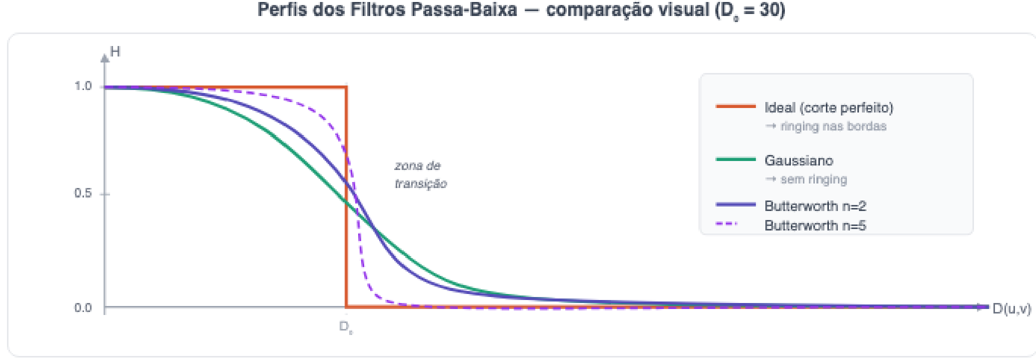


Figura 5.10: Filtros passa-baixa.

5.5.2 Filtros Passa-Alta e Passa-Banda

Filtros passa-alta são obtidos pelo complemento de qualquer filtro passa-baixa: $H_{\text{HP}} = 1 - H_{\text{LP}}$. O efeito visual é o inverso — preserva bordas e detalhes, atenua regiões uniformes.

Filtros passa-banda combinam um corte inferior D_L e um corte superior D_H, preservando apenas as frequências entre eles: $H_{\text{BP}} = H_{\text{HP}}^{(D_L)} \cdot H_{\text{LP}}^{(D_H)}$.

São especialmente úteis na **remoção de ruído periódico**: padrões regulares (interferência elétrica, grade de scanner) aparecem como picos no espectro de magnitude e podem ser suprimidos com um filtro rejeita-banda (*notch filter*) posicionado sobre esses picos. Ver exemplos de aplicações de filtros passa baixa na Figure 5.12 e passa alta na Figure 5.13.

Figura 5.11: Simulador interativo de filtros no domínio da frequência.

```
import io

def distancia_centro(M, N):
    """Matriz de distâncias ao centro do espectro."""
    u = np.arange(M) - M // 2
    v = np.arange(N) - N // 2
    V, U = np.meshgrid(v, u)
    return np.sqrt(U**2 + V**2)

def aplicar_filtro_freq(img, H):
    """Aplica filtro H (centrado) a imagem via FFT."""
    F  = np.fft.fftshift(np.fft.fft2(img.astype(np.float64)))
    Fg = F * H
    g  = np.real(np.fft.ifft2(np.fft.ifftshift(Fg)))
    return cv2.normalize(g, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

M, N = img_gray.shape
D    = distancia_centro(M, N)
D0   = 30      # frequência de corte
n_bw = 2       # ordem do Butterworth

#   Funções de transferência
H_ideal = (D <= D0).astype(np.float64)
H_gauss = np.exp(-D**2 / (2 * D0**2))
H_bw    = 1.0 / (1.0 + (D / D0)**(2 * n_bw))

#   Imagens filtradas
img_ideal = aplicar_filtro_freq(img_gray, H_ideal)
img_gauss = aplicar_filtro_freq(img_gray, H_gauss)
img_bw    = aplicar_filtro_freq(img_gray, H_bw)

#   Perfis de H(u,v)
def fig2img(fig):
    b = io.BytesIO(); fig.savefig(b, format='png', dpi=100); plt.close(fig); b.seek(0)
    return (plt.imread(b)[:,:,:3]*255).astype(np.uint8)

fig, ax = plt.subplots(figsize=(6, 3))
linha   = M // 2
ax.plot(H_ideal[linha, :], label="Ideal",       color="#D85A30", lw=1.5, ls="--")
ax.plot(H_gauss[linha, :], label="Gaussiano",   color="#1D9E75", lw=1.5)
ax.plot(H_bw[linha, :],    label="Butterworth", color="#534AB7", lw=1.5)
ax.axvline(N//2-D0, color="#aaa", lw=0.8, ls=":")
ax.axvline(N//2+D0, color="#aaa", lw=0.8, ls=":")
ax.set(title="Perfis H(u,v) - linha central", xlabel="v", ylabel="H(u,v)")
ax.legend(fontsize=8); plt.tight_layout()
perfil_img = fig2img(fig)

#   Filtros H visualizados
def H_vis(H):
    return cv2.normalize((H*255).astype(np.uint8), None, 0, 255, cv2.NORM_MINMAX)

mm.show(
    [img_gray, img_ideal, img_gauss, img_bw,
     H_vis(H_ideal), H_vis(H_gauss), H_vis(H_bw), perfil_img],
    titles=[
```

```
        "Original", "LPF Ideal", "LPF Gaussiano", "LPF Butterworth (n=2)",
        "H Ideal",  "H Gaussiano","H Butterworth", "Perfis H(u,v)"
    ],
    cols=4, figsize=(16, 9)
)
```

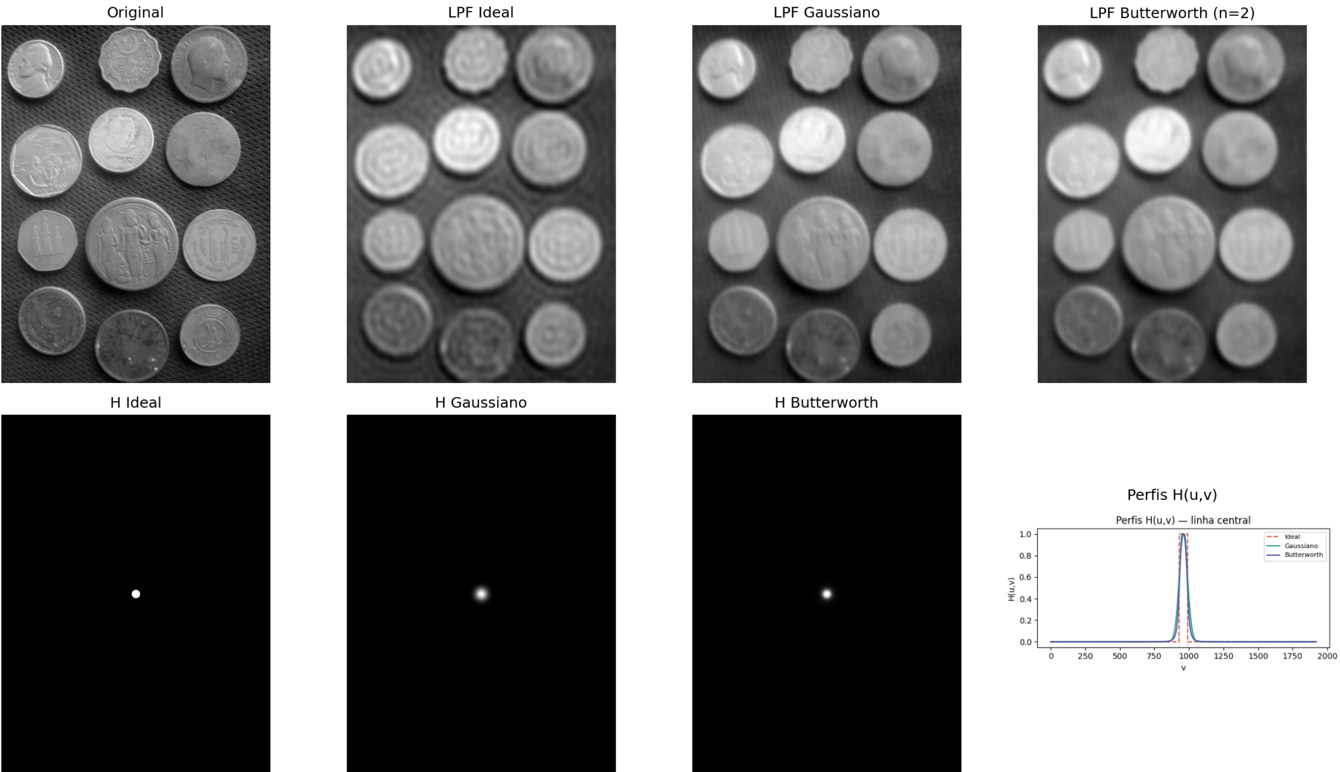


Figura 5.12: Comparação entre filtros passa-baixa: Ideal (D =30), Gaussiano (D =30) e Butterworth (D =30, n=2). Perfis de H(u,v) ao longo de uma linha central e imagens filtradas correspondentes.

```
# Filtro passa-alta: complemento do passa-baixa Gaussiano
# Reutiliza aplicar_filtro_freq() definida na célula anterior
H_alta   = 1 - H_gauss
img_alta = aplicar_filtro_freq(img_gray, H_alta)

plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1); plt.imshow(img_gray, cmap='gray'); plt.title('Original'); plt.axis('off')
plt.subplot(1, 2, 2); plt.imshow(img_alta, cmap='gray'); plt.title('Passa-alta Gaussiano (D =30)'); plt.
plt.tight_layout()
plt.show()
```

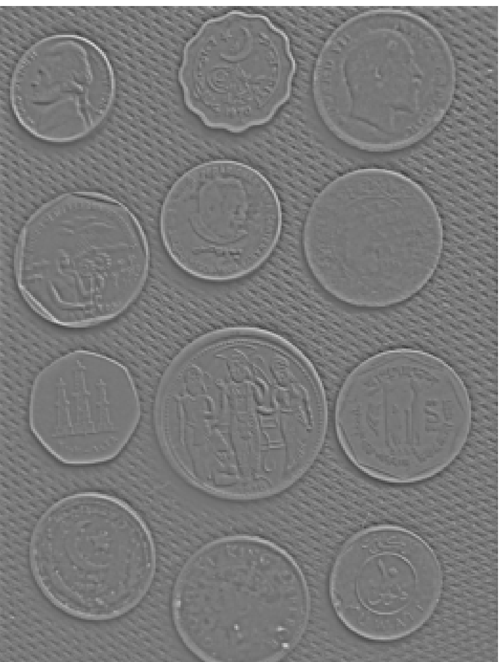


Figura 5.13: Filtro passa-alta Gaussiano. (a) Original; (b) Filtro passa-alta (D =30) - as bordas das moedas e fundo texturizado são realçados.

5.5.3 Remoção de Ruído Periódico

Ruído periódico — proveniente de interferência elétrica, sensores com padrão regular ou artefatos de digitalização — manifesta-se no espectro de Fourier como **picos pontuais** simétricos em torno do centro. O filtro **rejeita-banda notch** suprime seletivamente essas frequências, preservando o restante da imagem.

```
#   Imagem com ruído periódico sintético
h_img, w_img = img_gray.shape
x = np.arange(w_img); y = np.arange(h_img)
X, Y = np.meshgrid(x, y)

# Ruído senoidal com frequências (u0=20, v0=20) e simétricas
ruido = 40 * np.sin(2 * np.pi * (20 * X / w_img + 20 * Y / h_img))
img_ruidosa = np.clip(img_gray.astype(np.float64) + ruido, 0, 255).astype(np.uint8)

#   Espectro da imagem ruidosa
F_r  = np.fft.fftshift(np.fft.fft2(img_ruidosa.astype(np.float64)))
mag_r = np.log1p(np.abs(F_r))
mag_vis = cv2.normalize(mag_r, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

#   Máscara notch (suprime os picos do ruído)
mascara = np.ones((h_img, w_img), dtype=np.float64)
r_notch  = 10      # raio da região suprimida

def suprimir_pico(mask, cy, cx, r):
    """Zera um disco de raio r centrado em (cy, cx) na máscara."""
    yy, xx = np.ogrid[:mask.shape[0], :mask.shape[1]]
    dist    = np.sqrt((yy - cy)**2 + (xx - cx)**2)
    mask[dist <= r] = 0
    return mask

# Coordenadas dos picos no espectro centralizado
```

```
cy0, cx0 = h_img // 2, w_img // 2
dy0 = int(round(20 * h_img / h_img))   # = 20
dx0 = int(round(20 * w_img / w_img))   # = 20

for dy, dx in [(+dy0,+dx0),(-dy0,-dx0),(+dy0,-dx0),(-dy0,+dx0)]:
    mascara = suprimir_pico(mascara, cy0+dy, cx0+dx, r_notch)

mascara_vis = (mascara * 255).astype(np.uint8)

#   Filtragem e reconstrução
F_filtrada  = F_r * mascara
img_rest     = np.real(np.fft.ifft2(np.fft.ifftshift(F_filtrada)))
img_rest_vis = cv2.normalize(img_rest, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

psnr = cv2.PSNR(img_gray, img_rest_vis)
print(f"PSNR (original vs restaurada): {psnr:.2f} dB")

mm.show(
    [img_ruidosa, mag_vis, mascara_vis, img_rest_vis],
    titles=[
        "Com ruído periódico",
        "Espectro (log)",
        "Máscara notch",
        f"Restaurada (PSNR={psnr:.1f} dB)"
    ],
    cols=4, figsize=(16, 4)
)
```

```
PSNR (original vs restaurada): 30.62 dB
```

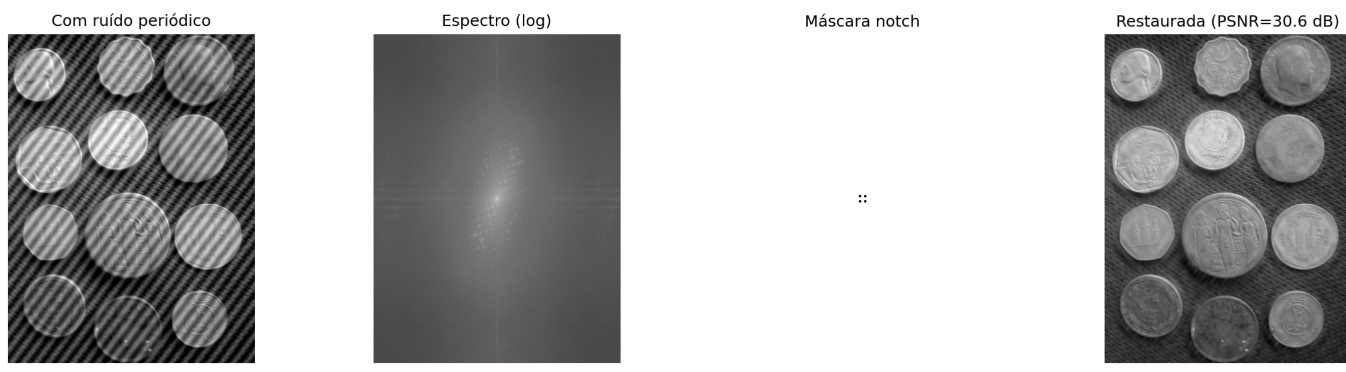


Figura 5.14: Remoção de ruído periódico via filtro notch no domínio da frequência: (a) imagem com ruído senoidal, (b) espectro mostrando os picos do ruído, (c) máscara notch centrada nos picos, (d) imagem restaurada.

📌 Síntese — Filtros Espectrais

Filtro	Efeito visual	Artefato	Uso
Passa-baixa Ideal	Suavização forte	Ringing (anéis)	Ilustrativo apenas
Passa-baixa Gaussiano	Suavização suave	Nenhum	Suavização geral
Passa-baixa Butterworth	Suavização controlada	Ringing leve (ordens altas)	Compromisso suavidade/precisão
Passa-alta	Realce de bordas	Pode amplificar ruído	Detecção de contornos
Notch	Remove frequências específicas	Pode criar artefatos locais	Remoção de ruído periódico

O design de filtros no domínio da frequência é direto e intuitivo — mas os artefatos espaciais (ringing, borramento) emergem de escolhas no espectro.

5.6 *Wavelets* e Multirresolução

A Transformada de Fourier decompõe o sinal em frequências **globais**: cada coeficiente $F(u, v)$ recebe contribuições de **toda** a imagem, sem informação sobre *onde* uma frequência ocorre. Uma borda localizada no canto da imagem se mistura ao espectro inteiro — o que limita sua capacidade de representar explicitamente onde determinadas frequências ocorrem espacialmente.

As ***wavelets*** (*ondaletas*) resolvem essa limitação com funções de base **compactas** — energia concentrada em uma região finita — que podem ser deslocadas e escaladas, oferecendo representação simultânea em **frequência e localização espacial**.

5.6.1 O Limite de Fourier: Onde ocorreu o evento?

Fourier nos diz perfeitamente *quais* frequências existem, mas perde totalmente a noção do *tempo/espaço* de onde elas ocorreram. Veja o experimento abaixo: uma imagem com duas linhas nítidas, e outra com as mesmas linhas deslocadas. O espectro de magnitude é virtualmente idêntico.

```
img_sinal1 = np.zeros((128, 128)); img_sinal1[:, 20:25] = 1; img_sinal1[100:105, :] = 1
img_sinal2 = np.zeros((128, 128)); img_sinal2[:, 90:95] = 1; img_sinal2[30:35, :] = 1

mag1 = np.log1p(np.abs(np.fft.fftshift(np.fft.fft2(img_sinal1))))
mag2 = np.log1p(np.abs(np.fft.fftshift(np.fft.fft2(img_sinal2))))

mm.show([img_sinal1, mag1, img_sinal2, mag2], titles=["Sinal A", "Espectro A", "Sinal B (Deslocado)", "E
```

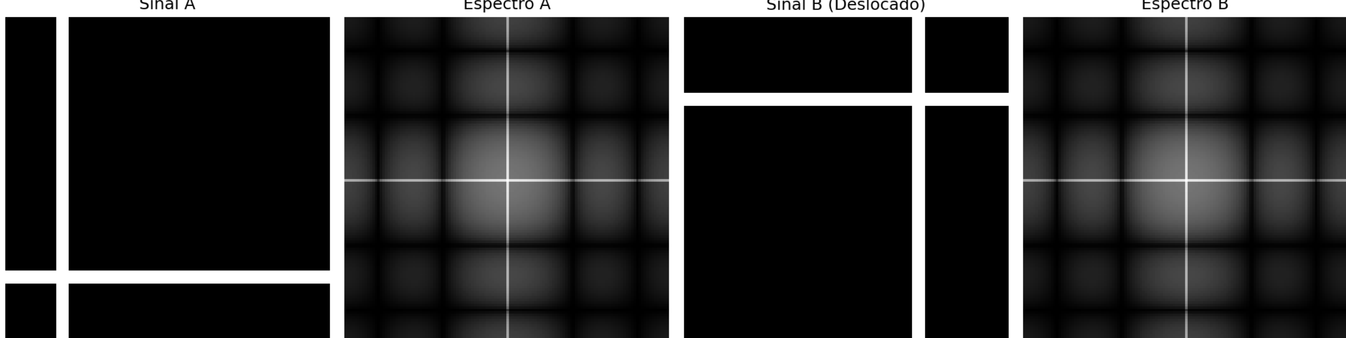


Figura 5.15: Fourier global é cego para a posição. Os espectros não dizem onde as bordas estão.

5.6.2 Transformada Wavelet Discreta 2D

A DWT aplica, separadamente em linhas e colunas, dois filtros complementares — **passa-baixa** h (aproximações) e **passa-alta** g (detalhes) — seguidos de subamostragem por 2, produzindo quatro subbandas:

$$\text{DWT}(f) = \{ \underbrace{\text{LL}}_{\text{aprox.}}, \underbrace{\text{LH}}_{\text{horiz.}}, \underbrace{\text{HL}}_{\text{vert.}}, \underbrace{\text{HH}}_{\text{diag.}} \}$$

Subbanda	Filtros aplicados	Conteúdo visual
LL	baixa × baixa	Aproximação suavizada — versão reduzida da imagem
LH	baixa × alta	Bordas horizontais e variações verticais

Subbanda	Filtros aplicados	Conteúdo visual
HL	alta × baixa	Bordas verticais e variações horizontais
HH	alta × alta	Detalhes diagonais, texturas em 45°, cantos

A decomposição é **recursiva**: aplicando a DWT novamente sobre LL obtém-se o próximo nível. Após J níveis, a representação hierárquica contém $3J + 1$ subbandas — cada nível com metade da resolução do anterior.

i Conexão com CNNs

A análise multirresolução das *Wavelets* possui forte relação conceitual com representações hierárquicas utilizadas em métodos modernos de visão computacional, como as CNNs.

5.6.3 Famílias de Wavelets

Diferentes *wavelets* resultam em diferentes compromissos entre localização, suavidade e compactação:

Wavelet	Suporte	Momentos nulos	Simetria	Uso típico
Haar	2	1	Assimétrica	Análise básica, educação
Daubechies db4	8	4	Assimétrica	Compressão, análise geral
Symlet sym4	8	4	Quase simétrica	Reconstrução de sinais
Biortogonal 5/3	5/3	2/2	Simétrica	JPEG 2000 (sem perda)
Biortogonal 9/7	9/7	4/4	Simétrica	JPEG 2000 (com perda)

O número de **momentos nulos** controla a compactação: com p momentos nulos, a *wavelet* produz coeficientes exatamente nulos para polinômios de grau até $p - 1$ — quanto maior p, mais coeficientes próximos de zero em regiões suaves e maior a eficiência de compressão.

```
wavelet_haar = pywt.Wavelet('haar')
wavelet_db4  = pywt.Wavelet('db4')

phi_h, psi_h, x_h = wavelet_haar.wavefun(level=4)
phi_d, psi_d, x_d = wavelet_db4.wavefun(level=4)

fig, ax = plt.subplots(1, 2, figsize=(10, 3))
ax[0].plot(x_h, psi_h, 'b', lw=2); ax[0].set_title("Ondaleta Haar ( )")
ax[1].plot(x_d, psi_d, 'g', lw=2); ax[1].set_title("Ondaleta Daubechies 4 ( )")
plt.tight_layout(); plt.show()
```

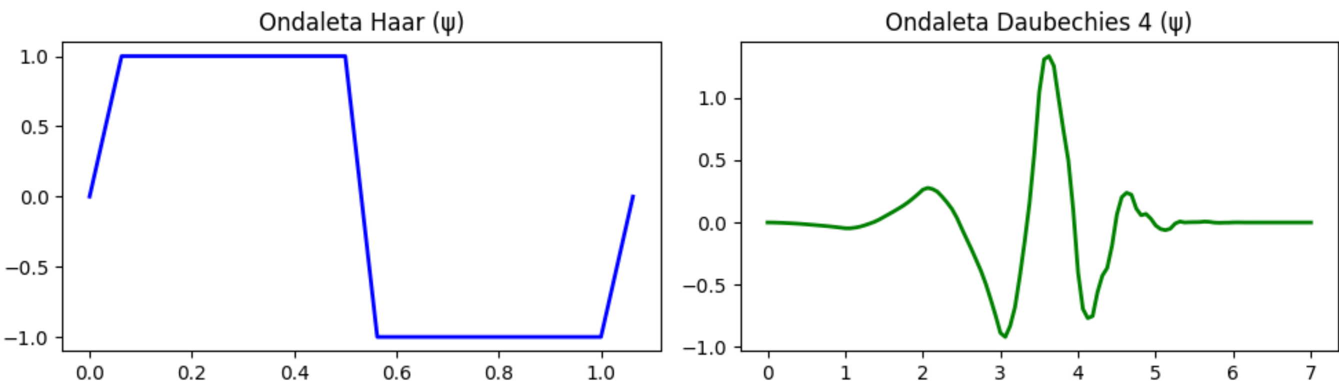


Figura 5.16: Funções da Wavelet ( ). Note como elas rapidamente decaem para zero (suporte compacto), ao contrário das senoides infinitas de Fourier.

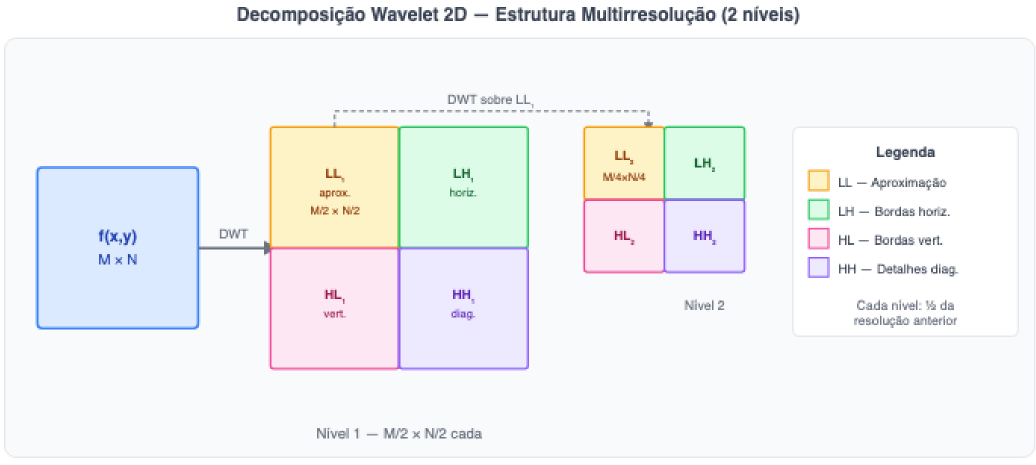
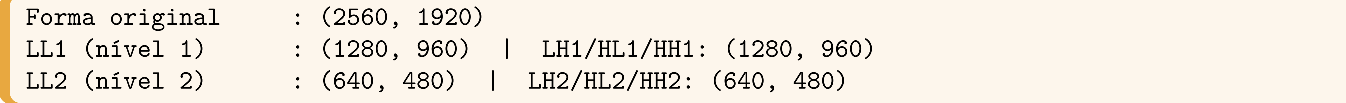


Figura 5.17: Diagrama da decomposição wavelet 2D em dois níveis.

```
try:
    import pywt
    HAS_PYWT = True
except ImportError:
    import subprocess
    subprocess.run(["pip", "install", "PyWavelets", "-q"])
    import pywt
    HAS_PYWT = True

#   Decomposição wavelet 2 níveis
wavelet   = "haar"
```

```
img_float = img_gray.astype(np.float64)

# Nível 1
coefs1 = pywt.dwt2(img_float, wavelet)
LL1, (LH1, HL1, HH1) = coefs1

# Nível 2 (aplicado sobre LL1)
coefs2 = pywt.dwt2(LL1, wavelet)
LL2, (LH2, HL2, HH2) = coefs2

def sb_vis(sb):
    """Normaliza subbanda para visualização [0,255]."""
    return cv2.normalize(np.abs(sb), None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

print(f"Forma original     : {img_gray.shape}")
print(f"LL1 (nível 1)      : {LL1.shape}  |  LH1/HL1/HH1: {LH1.shape}")
print(f"LL2 (nível 2)      : {LL2.shape}  |  LH2/HL2/HH2: {LH2.shape}")

imgs_dwt   = [img_gray, sb_vis(LL1), sb_vis(LH1), sb_vis(HL1), sb_vis(HH1),
              sb_vis(LL2), sb_vis(LH2), sb_vis(HL2), sb_vis(HH2)]
titles_dwt = ["Original",
              "LL (aprox.)", "LH (horiz.)", "HL (vert.)", "HH (diag.)",
              "LL (aprox.)", "LH (horiz.)", "HL (vert.)", "HH (diag.)"]

mm.show(imgs_dwt, titles=titles_dwt, cols=5, figsize=(16, 7))
```



```
Forma original     : (2560, 1920)
LL1 (nível 1)      : (1280, 960)  |  LH1/HL1/HH1: (1280, 960)
LL2 (nível 2)      : (640, 480)  |  LH2/HL2/HH2: (640, 480)
```

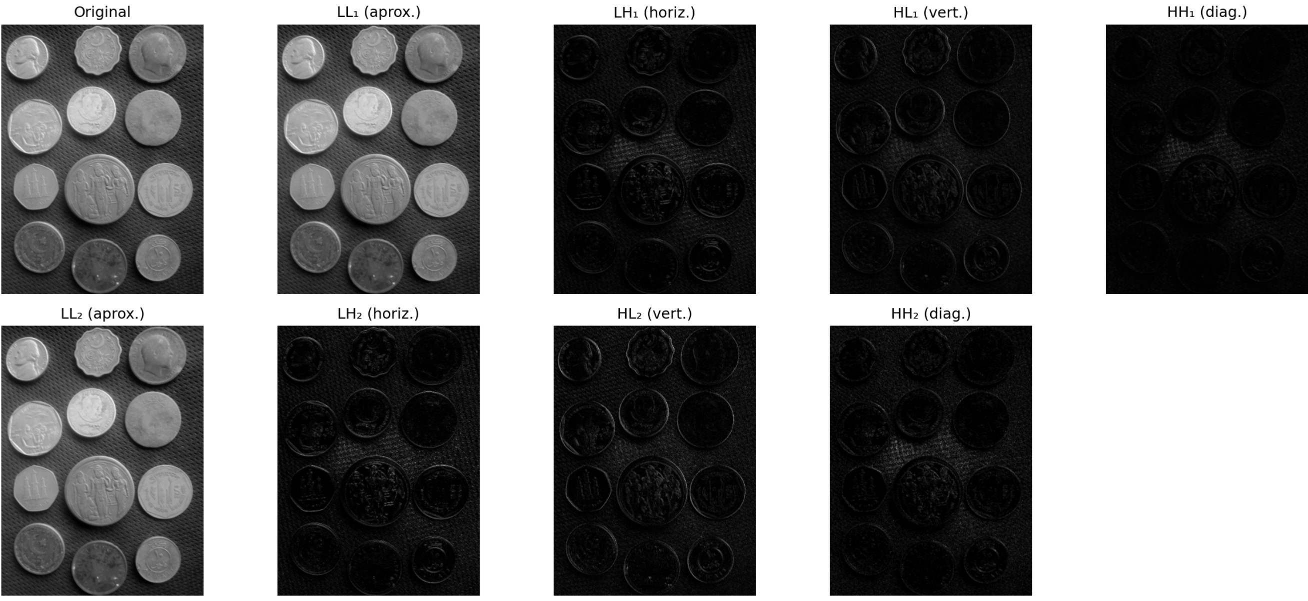


Figura 5.18: Decomposição wavelet 2D de 2 níveis com wavelet Haar: subbandas LL, LH, HL, HH em cada nível. As subbandas de detalhe revelam estruturas orientadas em diferentes escalas.

```
wavelets_comp = ["haar", "db4", "sym4", "bior2.2"]
imgs_comp, titles_comp = [], []

for wname in wavelets_comp:
    LL, (LH, HL, HH) = pywt.dwt2(img_float, wname)
    imgs_comp  += [sb_vis(LL), sb_vis(HH)]
```

```
    titles_comp += [f"{wname} - LL ", f"{wname} - HH "]

mm.show(imgs_comp, titles=titles_comp, cols=4, figsize=(14, 8))
```

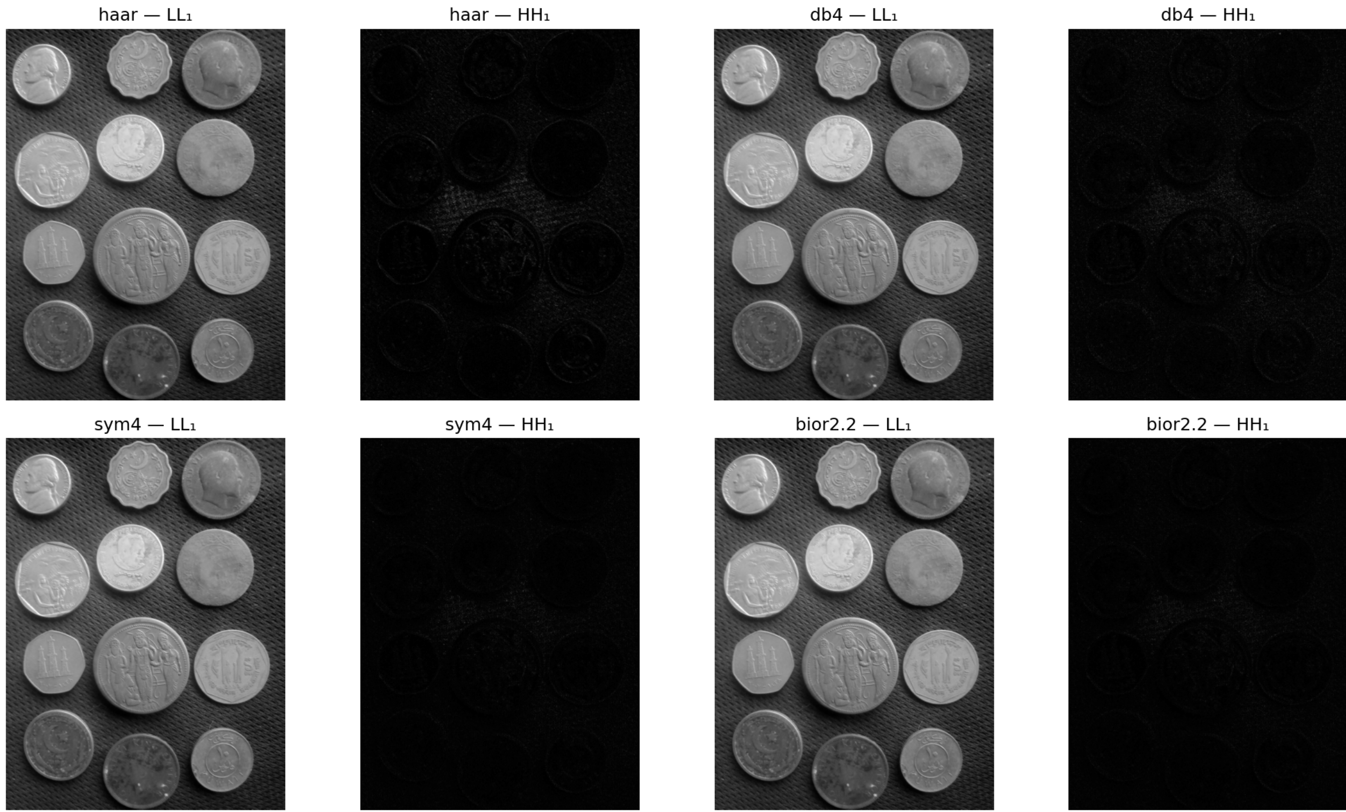


Figura 5.19: Comparação entre famílias de wavelets: Haar, db4, sym4 e bior2.2. Subbanda LL (aproximação) e HH (diagonal) para cada escolha, ilustrando o compromisso entre compactação e suavidade.

```
def dwt_threshold_reconstruct(img, wavelet='db4', nivel=2, threshold=0.0):
    """Decompõe, aplica limiar e reconstrói via IDWT."""
    coefs = pywt.wavedec2(img.astype(np.float64), wavelet, level=nivel)
    # Copia e aplica hard thresholding em todos os detalhe
    coefs_t = [coefs[0]]   # LL final não é limiarizado
    for detalhe in coefs[1:]:
        coefs_t.append(tuple(pywt.threshold(sb, threshold, mode='hard') for sb in detalhe))
    rec = pywt.waverec2(coefs_t, wavelet)
    # Recorte para dimensão original
    rec = rec[:img.shape[0], :img.shape[1]]
    return np.clip(rec, 0, 255).astype(np.uint8)

thresholds = [0, 10, 30, 60, 100]
imgs_thr   = [img_gray]
titles_thr = ["Original"]

for t in thresholds:
    rec  = dwt_threshold_reconstruct(img_gray, threshold=t)
    psnr = cv2.PSNR(img_gray, rec)
    imgs_thr  .append(rec)
    titles_thr.append(f"T={t}  PSNR={psnr:.1f} dB")

mm.show(imgs_thr, titles=titles_thr, cols=3, figsize=(14, 10))
```

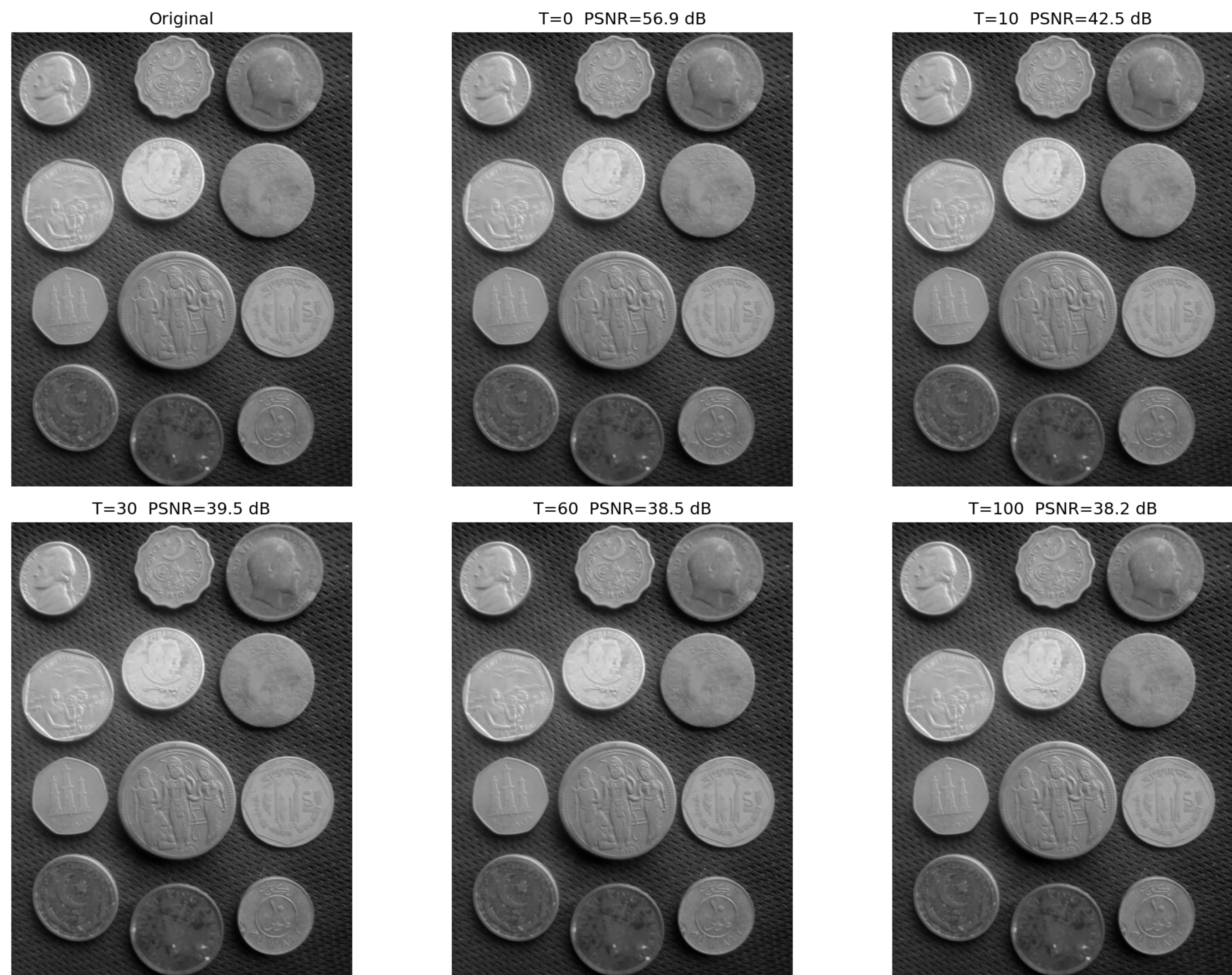


Figura 5.20: Reconstrução wavelet com limiamento de coeficientes (hard thresholding): à medida que o limiar aumenta, mais detalhes são zerificados, produzindo imagens progressivamente mais suaves. Métrica PSNR quantifica a perda de qualidade.

5.6.4 📌 Síntese - Fourier vs *Wavelets*: quando usar cada uma?

Critério	Fourier (DFT)	*Wavelet* (DWT)
Base	Senoides de suporte infinito	Funções compactas, localizadas
Localização espacial	— informação global	✓ — frequência *e* posição
Filtragem espectral	✓ — controle preciso de banda	Limitado
Compressão	DCT (JPEG) — excelente em blocos	DWT (JPEG 2000) — melhor para imagens inteiras
Análise multi-escala		✓ — hierarquia natural
Ruído periódico	✓ — localização exata no espectro	
Texturas não-estacionárias		✓

Regra prática: use Fourier para filtragem espectral e remoção de ruído periódico; use *wavelets* para compressão, análise multi-escala e processamento de sinais não-estacionários.

5.7 Compressão de Imagens

As wavelets estabelecem a fundação teórica do padrão JPEG 2000, mas o padrão JPEG original — dominante em fotografia digital — utiliza uma transformada mais simples e igualmente eficaz: a **Transformada de Cossenos Discreta (DCT)**. Ambas exploram o mesmo princípio: concentrar a energia da imagem em poucos coeficientes e descartar os demais com impacto visual mínimo.

A compressão visa reduzir a quantidade de dados necessária para armazenar ou transmitir uma imagem, explorando **redundâncias** presentes na representação original.

5.7.1 Taxonomia das Redundâncias

Três categorias principais de redundância são exploradas por algoritmos de compressão:

Tabela 5.7: Tipos de redundância em imagens e como são exploradas.

Tipo	Definição	Explorada por
Espacial (interpixel)	Pixels vizinhos são altamente correlacionados	DCT, codificação preditiva
Espectral (interchannel)	Canais de cor são correlacionados (R G B em imagens naturais)	Conversão YCbCr
Psicovisual	O sistema visual humano é insensível a certas variações de alta freq.	Quantização JPEG

A compressão é classificada em duas categorias fundamentais:

- **Sem perda (*lossless*):** reconstrução bit-a-bit idêntica ao original. Usada quando integridade é crítica (imagens médicas, documentos).
- **Com perda (*lossy*):** permite distorção controlada para ganhos maiores de compressão. Adequada para fotografia e vídeo, onde o SVH tolera imperfeições.

5.7.2 Transformada de Cossenos Discreta (DCT)

A **Transformada de Cossenos Discreta** (DCT) é a operação central do padrão JPEG. Diferentemente da DFT (que usa base complexa), a DCT é puramente real, o que a torna computacionalmente mais eficiente em implementações de propósito geral e a DCT é puramente real, o que simplifica implementações computacionais e favorece aplicações de compressão de imagens.

Para um bloco $f(x, y)$ de dimensões $N \times N$, a **DCT-II 2D** é definida como:

$$C(u, v) = \alpha(u)\,\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y)\, \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right] \tag{5.9}$$

onde $\alpha(0) = \sqrt{1/N}$ e $\alpha(k) = \sqrt{2/N}$ para $k > 0$ (fator de normalização ortogonal). O coeficiente $C(0, 0)$ corresponde ao **componente DC**.

DCT vs DFT: vantagem da compactação de energia

Tanto a DCT quanto a DFT transformam um bloco $N \times N$ em $N \times N$ coeficientes. A diferença crítica para compressão é a **compactação de energia**: para imagens naturais. Para imagens naturais, a DCT frequentemente apresenta melhor compactação de energia nos coeficientes de baixa frequência do que a DFT, favorecendo aplicações de compressão perceptual. Isso ocorre porque a DCT assume simetria par do sinal — equivalente a uma extensão periódica sem descontinuidade — reduzindo o

ringing nas bordas do bloco. A consequência prática é que mais coeficientes DCT podem ser zerados sem degradação visível, resultando em maior compressão.

```
fig, axes = plt.subplots(8, 8, figsize=(6, 6))
fig.subplots_adjust(hspace=0.05, wspace=0.05)
for i in range(8):
    for j in range(8):
        # Criamos um "impulso" isolado na frequência
        coef = np.zeros((8, 8)); coef[i, j] = 1
        # A inversa revela o padrão geométrico espacial equivalente
        b = idct(idct(coef.T, norm='ortho').T, norm='ortho')
        axes[i, j].imshow(b, cmap='gray')
        axes[i, j].axis('off')
plt.suptitle("As 64 Bases da DCT 8x8", y=0.92, fontsize=12, fontweight='bold')
plt.show()
```

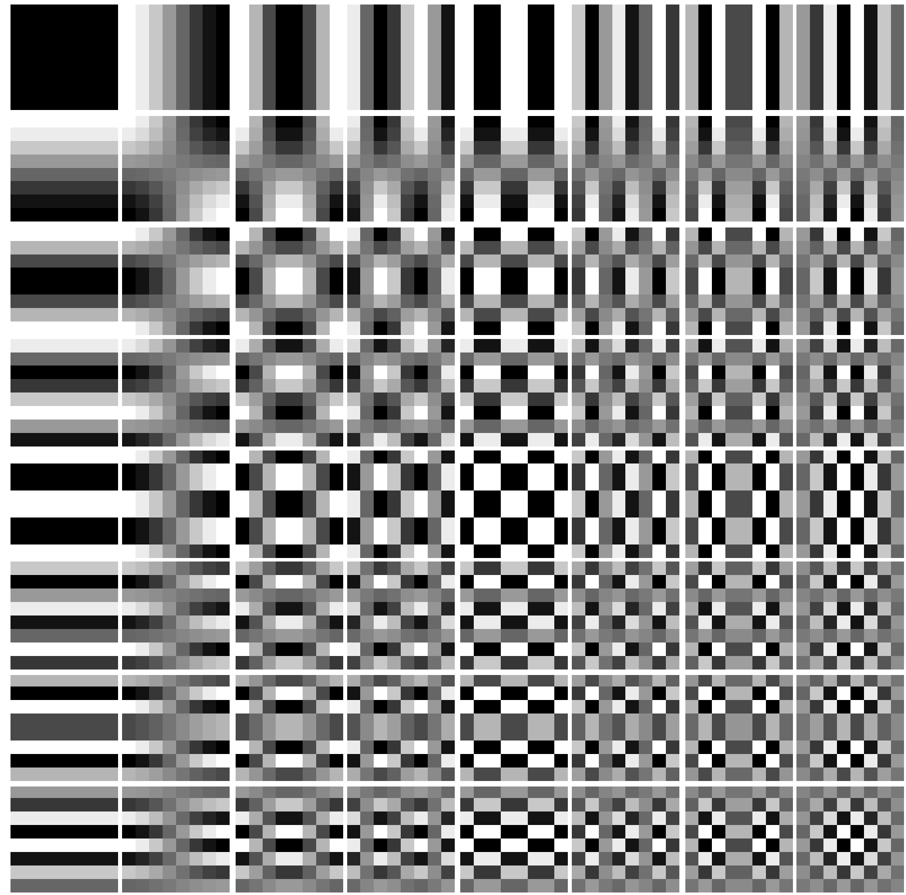


Figura 5.21: O Alfabeto Visual do JPEG: As 64 funções de base da DCT-II. O coeficiente DC fica no topo esquerdo (suave). Ao descer e avançar à direita, a oscilação espacial aumenta drasticamente.

5.7.3 Quantização: a principal fonte de compressão

Após a DCT, cada coeficiente $C(u,v)$ é dividido por um valor da **tabela de quantização** $Q(u,v)$ e arredondado para o inteiro mais próximo. Coeficientes de alta frequência — menos perceptíveis ao SVH — recebem valores altos em Q, forçando o resultado ao arredondamento para **zero**. Longas sequências de zeros são então comprimidas eficientemente pela codificação entrópica.

Quando o fator de qualidade é baixo, o descarte de altas frequências é agressivo: o bloco 8×8 é reconstruído

com poucos coeficientes, o que produz os característicos **artefatos de bloco** visíveis nas fronteiras entre blocos adjacentes.

```python
from scipy.fft import dct, idct

def dct2(bloco):
    """DCT-II 2D ortogonal (separável)."""
    return dct(dct(bloco.T, norm='ortho').T, norm='ortho')

def idct2(coefs):
    """IDCT-II 2D ortogonal."""
    return idct(idct(coefs.T, norm='ortho').T, norm='ortho')

#   Bloco 8×8 centralizado da imagem
cy, cx = img_gray.shape[0]//2, img_gray.shape[1]//2
bloco  = img_gray[cy:cy+8, cx:cx+8].astype(np.float64) - 128.0

C = dct2(bloco)

print("Coeficientes DCT do bloco 8×8:")
print(np.round(C).astype(int))
print(f"\nEnergia DC     : {C[0,0]**2:.1f}")
print(f"Energia total  : {(C**2).sum():.1f}")
print(f"Fração no DC   : {C[0,0]**2 / (C**2).sum():.1%} ← concentração de energia")

#   Reconstrução progressiva
imgs_rec   = [cv2.normalize((bloco+128).astype(np.uint8), None, 0, 255, cv2.NORM_MINMAX)]
titles_rec = ["Bloco original\n(8×8 pixels)"]

for keep in [1, 4, 10, 20, 40, 64]:
    C_trunc = np.zeros_like(C)
    indices = sorted([(u,v) for u in range(8) for v in range(8)], key=lambda p: p[0]+p[1])
    for u, v in indices[:keep]:
        C_trunc[u, v] = C[u, v]
    rec = np.clip(idct2(C_trunc) + 128, 0, 255).astype(np.uint8)
    imgs_rec.append(rec)
    titles_rec.append(f"{keep} coef.\n({keep/64:.0%} do total)")

mm.show(imgs_rec, titles=titles_rec, cols=4, figsize=(12, 7))
```

```
Coeficientes DCT do bloco 8×8:
[[192 -48   1   8   0  -1   0   0]
 [-96  54   7 -10   0   0   0   0]
 [ 14 -14   8   0   0   0   0   0]
 [  0   0 -11   1   0   0   0   0]
 [  9 -11   0   0   1   0   1   0]
 [  0   0   0   0   0   0   0   0]
 [  0   0   0   0  -1   0   0   0]
 [  0   0   0   0   0   0   0   0]]

Energia DC     : 36816.0
Energia total  : 52253.0
Fração no DC   : 70.5% ← concentração de energia
```

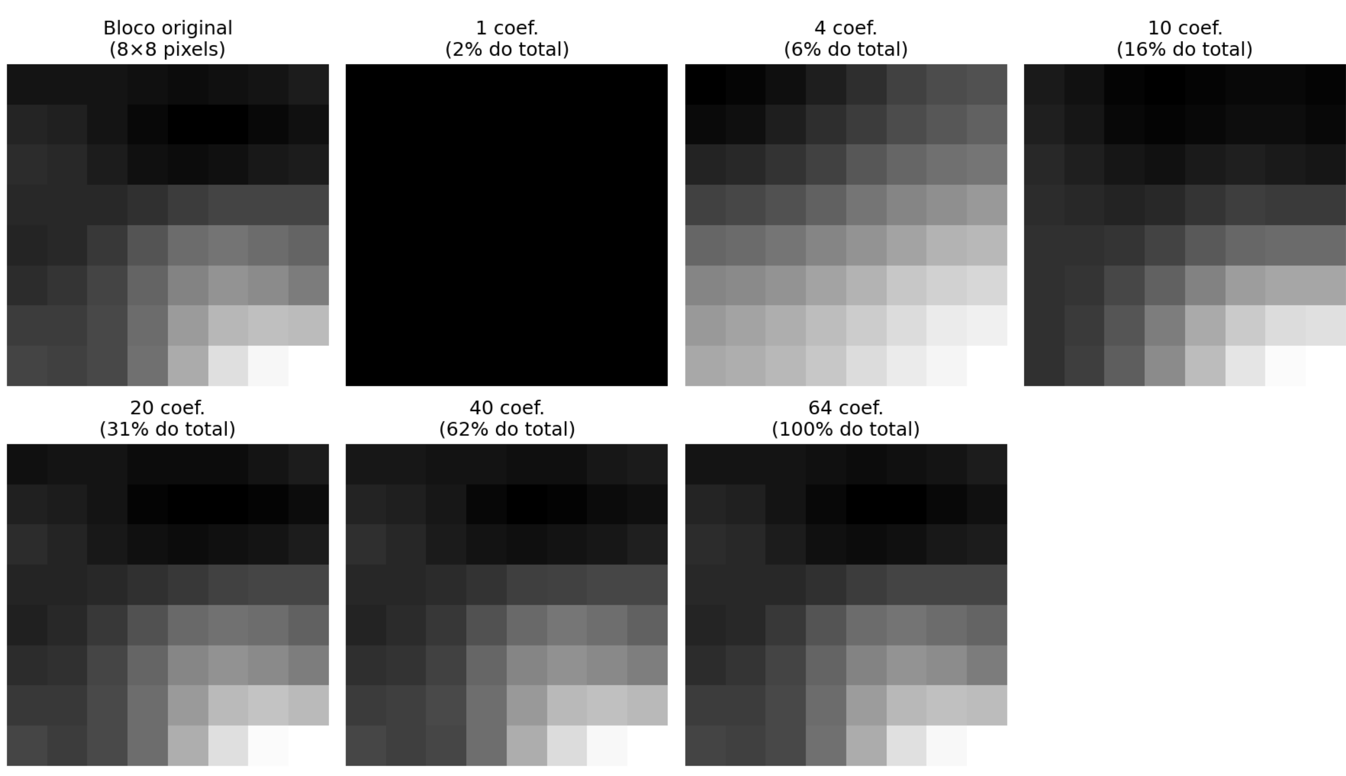


Figura 5.22: DCT 2D em bloco 8×8: coeficientes e reconstrução progressiva.

5.7.4 Pipeline JPEG

O padrão JPEG aplica a DCT em blocos disjuntos de 8×8 pixels. O pipeline completo envolve seis etapas principais:

$$\text{RGB} \xrightarrow{\text{(1) YCbCr}} \xrightarrow{\text{(2) Sub-amostragem 4:2:0}} \xrightarrow{\text{(3) Blocos 8×8}} \xrightarrow{\text{(4) DCT}} \xrightarrow{\text{(5) Quantização}} \xrightarrow{\text{(6) Huffman/RLE}}$$

Etapa	Operação	Fundamento perceptual
1	RGB → YCbCr: separa luminância (Y) de crominância (Cb, Cr)	O SVH é ~4× mais sensível a variações de luminância do que de cor
2	Sub-amostragem 4:2:0: Cb e Cr reduzidos à metade da resolução	Elimina ~50% dos dados de cor com impacto visual mínimo
3–4	Blocos 8×8 centrados em zero e transformados pela DCT	Concentra energia nos primeiros coeficientes
5	Divisão por $Q(u,v)$ e arredondamento: $\tilde{C}(u,v) = \text{round}[C(u,v)/Q(u,v)]$	Zeros em altas frequências → principal fonte de compressão
6	Varredura zigzag + RLE + Huffman	Comprime longas sequências de zeros resultantes da quantização

A **tabela de quantização** Q é o parâmetro central do compromisso qualidade-compressão: o fator de qualidade JPEG (1–100) escala Q globalmente — valores altos preservam mais coeficientes; valores baixos geram mais zeros e maiores artefatos.

Por que o Ziguezague? A DCT acumula a energia vital no canto superior esquerdo (baixas frequências) e empurra os zeros para a direita e para baixo. Ler a matriz na diagonal (ziguezague) cria uma "corrida de zeros" contínua no final do vetor, permitindo que o algoritmo simplesmente anote "e os próximos 45 elementos são zero", comprimindo o dado violentamente (*Run-Length Encoding*).

```
#   Tabela de quantização luminância (padrão JPEG)
Q_luma = np.array([
    [16,11,10,16,24,40,51,61],
    [12,12,14,19,26,58,60,55],
    [14,13,16,24,40,57,69,56],
    [14,17,22,29,51,87,80,62],
    [18,22,37,56,68,109,103,77],
    [24,35,55,64,81,104,113,92],
    [49,64,78,87,103,121,120,101],
    [72,92,95,98,112,100,103,99]
], dtype=np.float64)

def jpeg_compress_block(bloco, Q_table):
    """DCT → quantização → dequantização → IDCT em bloco 8×8."""
    C  = dct2(bloco.astype(np.float64) - 128)
    Cq = np.round(C / Q_table) * Q_table    # quantiza e dequantiza
    return np.clip(idct2(Cq) + 128, 0, 255)

def jpeg_quality_compress(img, qualidade=50):
    """JPEG simplificado: comprime imagem inteira por blocos 8×8."""
    # Fator de escala: qualidade 50 = sem escala; >50 = maior qualidade
    if qualidade < 50:
        escala = 5000 / qualidade
    else:
        escala = 200 - 2 * qualidade
    Q = np.clip(np.round(Q_luma * escala / 100), 1, 255)

    h, w   = img.shape
    result = np.zeros_like(img, dtype=np.float64)
    for r in range(0, h-7, 8):
        for c in range(0, w-7, 8):
            result[r:r+8, c:c+8] = jpeg_compress_block(img[r:r+8, c:c+8], Q)
    return result.astype(np.uint8)

#   Comparação de fatores de qualidade
qualidades = [10, 25, 50, 75, 90]
imgs_jpeg  = [img_gray]
titles_jpeg = ["Original"]

for q in qualidades:
    rec  = jpeg_quality_compress(img_gray, qualidade=q)
    psnr = cv2.PSNR(img_gray, rec)
    imgs_jpeg  .append(rec)
    titles_jpeg.append(f"Q={q}  PSNR={psnr:.1f}dB")

mm.show(imgs_jpeg, titles=titles_jpeg, cols=3, figsize=(14, 10))
```

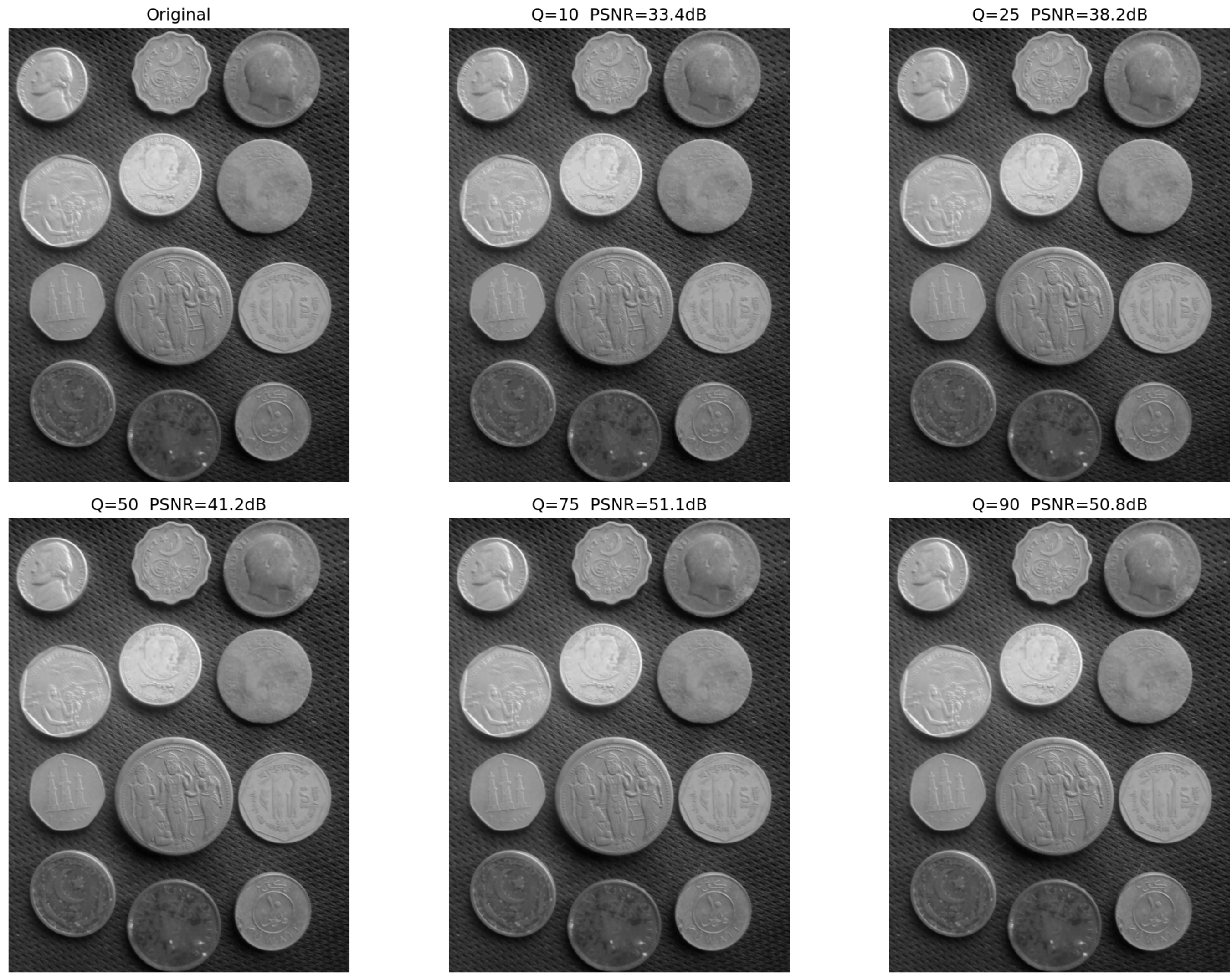


Figura 5.23: Pipeline JPEG simplificado: DCT em blocos 8×8, quantização com diferentes fatores de qualidade e reconstrução via IDCT. O artefato de blocos torna-se visível para qualidades baixas (Q=10–20).

5.7.5 Simulador Interativo: Quantização DCT

O simulador abaixo permite explorar o efeito da quantização sobre um bloco 8×8 real, visualizando em tempo real:

- os **coeficientes DCT** (mapa de calor — quanto maior, mais quente);
- os **coeficientes quantizados** (observe os zeros aparecerem com qualidade baixa);
- o **bloco reconstruído** e o erro de quantização.

5.8 Comparação de Formatos de Imagem

A escolha do formato de arquivo impacta diretamente o compromisso entre qualidade, tamanho e velocidade de decodificação. Os três formatos mais relevantes para aplicações web e computação visual são JPEG, PNG e WebP.

5.8.1 Características dos Formatos

Figura 5.24: Simulador interativo de compressão DCT-JPEG: ajuste o fator de qualidade e visualize em tempo real os coeficientes zerados, o bloco reconstruído e o erro de quantização.

Tabela 5.9: Comparação entre os principais formatos de imagem rasterizados.

Característica	JPEG	PNG	WebP
Compressão	Com perda	Sem perda	Com e sem perda
Transparência (alfa)		✓	✓
Suporte a animação		Limitado	✓
Algoritmo base	DCT + Huffman	DEFLATE (LZ77 + Huffman)	VP8 / VP8L
Melhor para	Fotografia	Gráficos, texto, ícones	Web (substituto universal)
Pior para	Texto, bordas nítidas	Fotos de alta resolução	Compatibilidade legada

5.8.2 Métricas de Qualidade

Duas métricas objetivas são amplamente usadas para avaliar a qualidade de imagens comprimidas:

PSNR (Peak Signal-to-Noise Ratio):

$$\text{PSNR} = 10 \log_{10}\left(\frac{L^2}{\text{MSE}}\right) \quad [\text{dB}] \tag{5.10}$$

onde $L = 255$ para imagens de 8 bits e MSE é o erro quadrático médio. Valores típicos: PSNR > 40 dB indica qualidade excelente; 30–40 dB, qualidade boa; < 30 dB, degradação visível.

SSIM (Structural Similarity Index):

$$\text{SSIM}(f, g) = \frac{(2\mu_f\mu_g + c_1)(2\sigma_{fg} + c_2)}{(\mu_f^2 + \mu_g^2 + c_1)(\sigma_f^2 + \sigma_g^2 + c_2)} \tag{5.11}$$

O SSIM combina três fatores medidos em janelas locais: **luminância** (μ_f, μ_g), **contraste** (σ_f, σ_g) e **estrutura** (σ_{fg}), ponderados por constantes de estabilidade c_1, c_2. O resultado varia em $[-1, 1]$, com 1 indicando identidade perfeita. Diferentemente do PSNR, o SSIM é sensível à organização espacial dos erros, sendo mais alinhado com a percepção humana — consulte Gonzalez; Woods (2018) para a formulação completa.

i PSNR vs SSIM: qual usar?

O PSNR é simples e rápido, mas pode superestimar a qualidade em imagens com artefatos localizados (blocos JPEG) ou subestimá-la quando o ruído é perceptualmente irrelevante (diferença de brilho global). O SSIM captura melhor a percepção humana, mas é mais caro computacionalmente.
Para avaliação rigorosa de algoritmos de compressão, recomenda-se reportar ambas as métricas, além de avaliação subjetiva em estudos perceptuais.

5.8.3 Inspeção Visual: Qualidade não é só PSNR

A filosofia de compressão dita o tipo de degradação. O JPEG corta a imagem em quadrados (DCT), gerando “Mosaicos”. Formatos baseados em wavelets ou filtros preditivos modernos (como WebP/JPEG2000) evitam blocos, mas sofrem de perda de textura e “derretimento” (borramento).

```python
# Recorte de área de alto contraste das imagens já comprimidas no loop anterior (Dando zoom de 4x)
zoom_original = cv2.resize(img_gray[120:200, 150:230], (320, 320), interpolation=cv2.INTER_NEAREST)

rec_jpeg = cv2.imread("imagens/comp_test/coins_q10.jpg", 0)
```

```
zoom_jpeg = cv2.resize(rec_jpeg[120:200, 150:230], (320, 320), interpolation=cv2.INTER_NEAREST)

# WebP Q=10
cv2.imwrite("imagens/comp_test/coins_q10.webp", img_gray, [cv2.IMWRITE_WEBP_QUALITY, 10])
rec_webp = cv2.imread("imagens/comp_test/coins_q10.webp", 0)
zoom_webp = cv2.resize(rec_webp[120:200, 150:230], (320, 320), interpolation=cv2.INTER_NEAREST)

mm.show([zoom_original, zoom_jpeg, zoom_webp], titles=["Zoom Original", "JPEG Q=10 (DCT Blocos)", "WebP
```

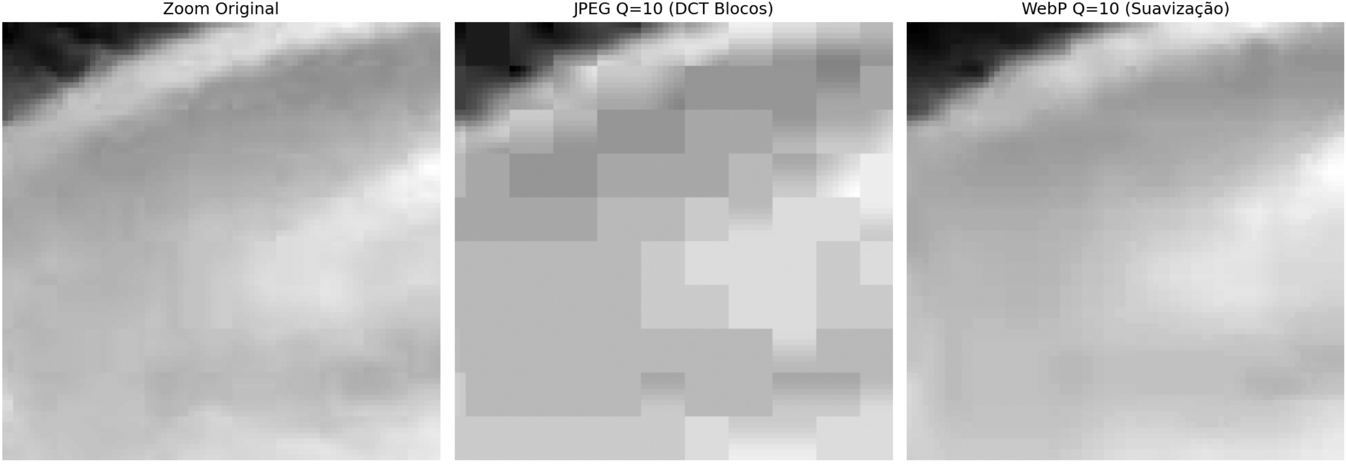


Figura 5.25: Forte compressão (Q=10). À esquerda o artefato de bloco clássico da DCT. À direita, a suavização de bordas característica do WebP.

```
os.makedirs("imagens/comp_test", exist_ok=True)
resultados = []

#   JPEG
for q in [10, 20, 30, 40, 50, 60, 70, 80, 90, 95]:
    path = f"imagens/comp_test/coins_q{q}.jpg"
    cv2.imwrite(path, img_gray, [cv2.IMWRITE_JPEG_QUALITY, q])
    rec  = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
    resultados.append({"formato": "JPEG", "qualidade": q,
                       "PSNR": cv2.PSNR(img_gray, rec),
                       "KB":   os.path.getsize(path)/1024})

#   PNG
path_png = "imagens/comp_test/coins.png"
cv2.imwrite(path_png, img_gray, [cv2.IMWRITE_PNG_COMPRESSION, 9])
resultados.append({"formato": "PNG", "qualidade": "lossless",
                   "PSNR": float('inf'), "KB": os.path.getsize(path_png)/1024})

#   WebP
for q in [50, 75, 90]:
    path_w = f"imagens/comp_test/coins_q{q}.webp"
    cv2.imwrite(path_w, img_gray, [cv2.IMWRITE_WEBP_QUALITY, q])
    rec_w  = cv2.imread(path_w, cv2.IMREAD_GRAYSCALE)
    resultados.append({"formato": "WebP", "qualidade": q,
                       "PSNR": cv2.PSNR(img_gray, rec_w),
                       "KB":   os.path.getsize(path_w)/1024})

#   Curva taxa-distorção
jpeg_r = [r for r in resultados if r["formato"]=="JPEG"]
webp_r = [r for r in resultados if r["formato"]=="WebP"]
png_r  = [r for r in resultados if r["formato"]=="PNG"]
```

```python
fig, ax = plt.subplots(figsize=(8, 4.5))
ax.plot([r["KB"] for r in jpeg_r], [r["PSNR"] for r in jpeg_r],
        "o-", label="JPEG", color="#D85A30", lw=2, ms=5)
ax.plot([r["KB"] for r in webp_r], [r["PSNR"] for r in webp_r],
        "s-", label="WebP", color="#534AB7", lw=2, ms=5)
ax.axhline(50, color="#1D9E75", lw=2, ls="--",
           label=f"PNG sem perda ({png_r[0]['KB']:.0f} KB)")
ax.axhspan(40, 60, alpha=0.05, color="#1D9E75", label="Qualidade excelente (PSNR>40)")
ax.set(xlabel="Tamanho do arquivo (KB)", ylabel="PSNR (dB)",
       title="Curva Taxa-Distorção: JPEG × WebP × PNG")
ax.legend(fontsize=9); ax.grid(True, alpha=0.3); plt.tight_layout()
plt.show()

print(f"\nTamanho bruto (sem compressão): {img_gray.nbytes/1024:.0f} KB")
print(f"\n{'Formato':>8} {'Qual.':>6} {'KB':>7} {'PSNR (dB)':>11}")
print("-"*38)
for r in resultados:
    psnr_s = f"{r['PSNR']:>11.2f}" if r['PSNR']!=float('inf') else f"{'∞ (lossless)':>11}"
    print(f"{r['formato']:>8} {str(r['qualidade']):>6} {r['KB']:>7.1f} {psnr_s}")
```

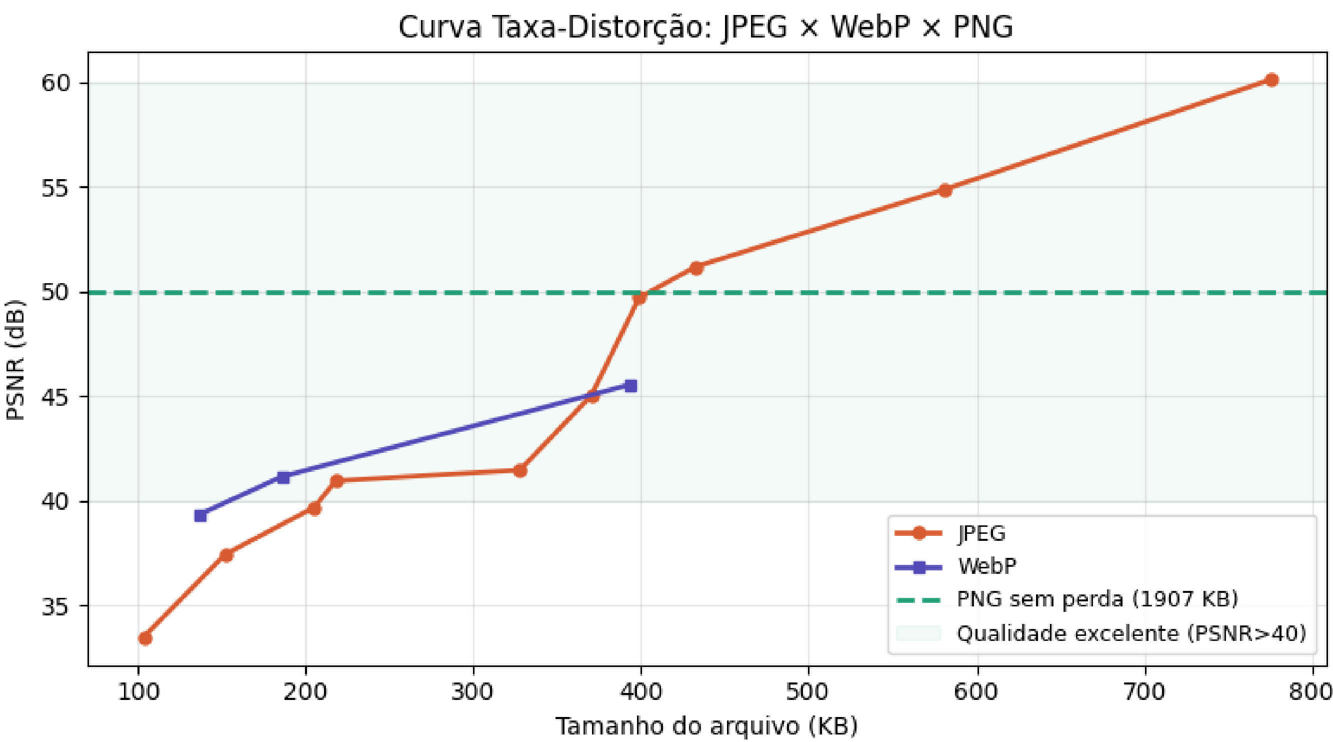


Figura 5.26: Curva taxa-distorção: PSNR vs tamanho de arquivo para JPEG, WebP e PNG.

```
Tamanho bruto (sem compressão): 4800 KB

 Formato  Qual.      KB   PSNR (dB)
--------------------------------------
    JPEG     10   103.3       33.50
    JPEG     20   152.0       37.45
    JPEG     30   205.1       39.70
    JPEG     40   217.8       40.96
    JPEG     50   327.7       41.48
    JPEG     60   370.4       45.04
    JPEG     70   398.8       49.71
    JPEG     80   432.4       51.19
```

```
    JPEG     90   581.0       54.88
    JPEG     95   775.3       60.12
     PNG lossless  1906.8 ∞ (lossless)
    WebP     50   137.0       39.37
    WebP     75   185.9       41.16
    WebP     90   393.1       45.55
```

```
try:
    from skimage.metrics import structural_similarity as ssim
except ImportError:
    import subprocess; subprocess.run(["pip","install","scikit-image","-q"])
    from skimage.metrics import structural_similarity as ssim

qualidades_ssim = [25, 50, 75, 95]
imgs_ssim   = [img_gray]
titles_ssim = ["Original"]

for q in qualidades_ssim:
    path = f"imagens/comp_test/coins_q{q}.jpg"
    rec  = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
    if rec is None: continue
    if rec.shape != img_gray.shape:
        rec = cv2.resize(rec, (img_gray.shape[1], img_gray.shape[0]))

    psnr_v = cv2.PSNR(img_gray, rec)
    ssim_v, _ = ssim(img_gray, rec, full=True)
    diff_vis = cv2.normalize(np.abs(img_gray.astype(float)-rec.astype(float)),
                             None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

    imgs_ssim  += [rec, diff_vis]
    titles_ssim += [f"Q={q}  PSNR={psnr_v:.1f}dB\nSSIM={ssim_v:.3f}",
                    f"Mapa de erro (Q={q})\n(bordas = artefatos de bloco)"]

mm.show(imgs_ssim, titles=titles_ssim, cols=3, figsize=(14, 14))
```

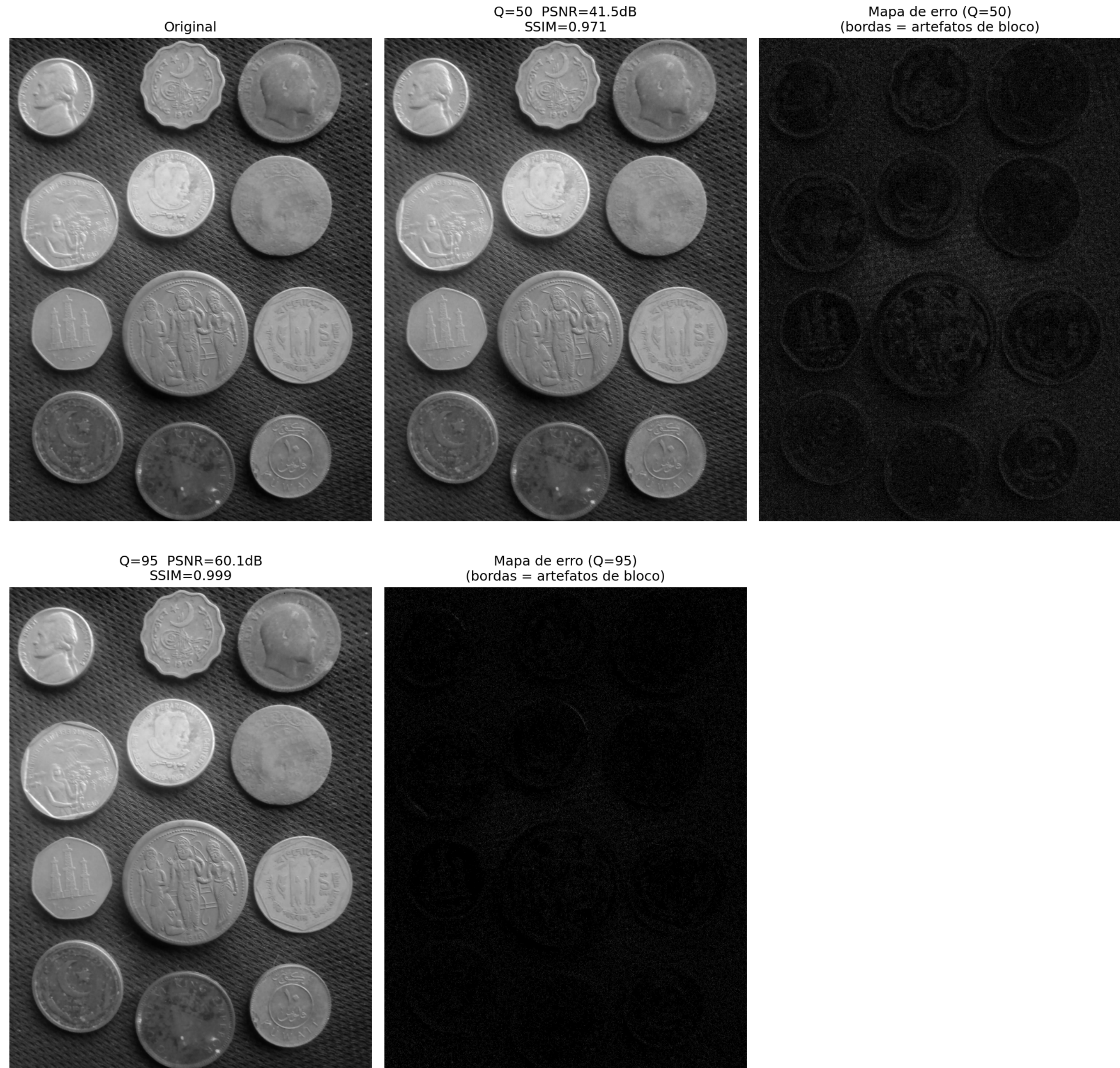


Figura 5.27: Mapas de erro JPEG em diferentes qualidades: artefatos de bloco nas bordas.

📌 Síntese — Compressão JPEG

Etapa	O que faz	Ganho
YCbCr	Separa luminância de crominância	Modela percepção humana
Subamostragem 4:2:0	Reduz Cb/Cr à metade	~50% de dados a menos
DCT 8×8	Concentra energia nos primeiros coeficientes	Alta compactação
Quantização	Zera coeficientes imperceptíveis	Principal fonte de compressão
Huffman	Codifica estatisticamente os zeros	~30–50% adicional

Artefatos típicos de JPEG: - **Artefato de bloco** (qualidades baixas): fronteiras 8×8 visíveis; - ***Ringing*** (qualidades muito baixas): oscilações ao redor de bordas nítidas; - **Perda de textura** (qualidades baixas):

regiões com textura fina ficam "plásticas".

5.9 Aplicação Prática: Remoção de Ruído por Filtragem Espectral

Reunindo as técnicas do capítulo, apresenta-se um *pipeline* completo de **remoção de ruído** que combina análise no domínio da frequência com filtragem adaptativa. O objetivo é remover um ruído misto (Gaussiano + periódico) preservando ao máximo a estrutura original da imagem.

$$\text{Ruidosa} \xrightarrow{\text{FFT}} \xrightarrow{\text{Identif. picos}} \xrightarrow{\text{Filtro Butterworth + Notch}} \xrightarrow{\text{IFFT}} \text{Restaurada}$$

Avaliação: PSNR e SSIM

O par de métricas PSNR / SSIM fornece uma avaliação complementar: - **PSNR** penaliza uniformemente todos os erros pixel a pixel. - **SSIM** avalia a preservação de estruturas perceptualmente relevantes. Um bom filtro maximiza ambos, mas na prática existe um compromisso: filtros muito agressivos reduzem o ruído de alta frequência mas destroem bordas e texturas, o que pode melhorar o PSNR em relação à imagem ruidosa mas reduz o PSNR em relação à imagem original e degrada o SSIM.

5.10 Resumo do Capítulo

A transição do **domínio espacial** para o **domínio da frequência** revela a distribuição de energia da imagem, estabelecendo a base matemática para filtragem avançada e compressão de dados. Ver um mapa conceitual do domínio da frequência na Figure 5.28.

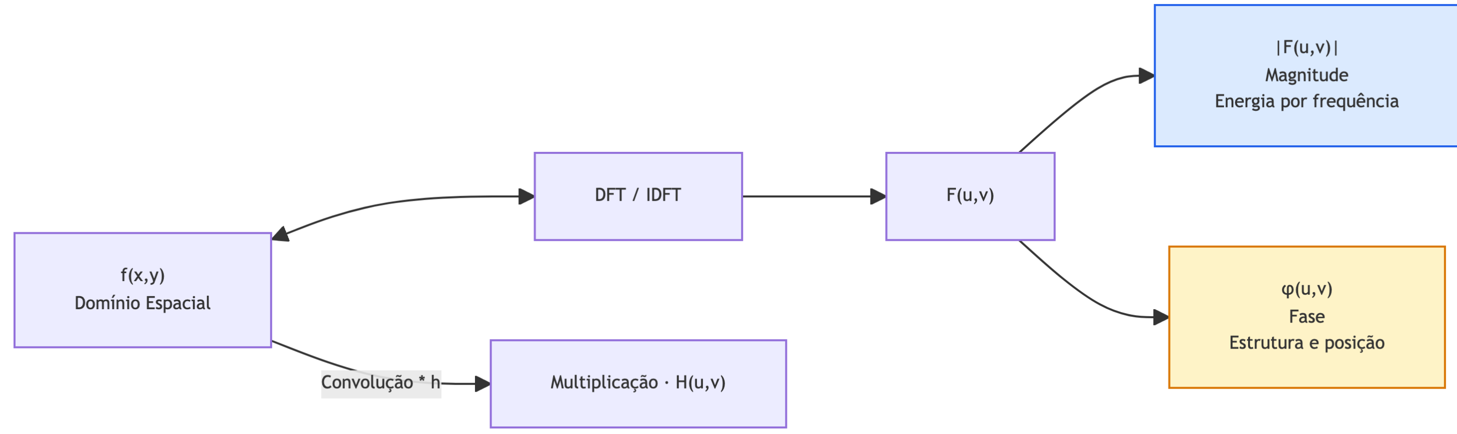


Figura 5.28: Mapa conceitual do domínio da frequência.

Fundamentos Essenciais

- **DFT e Percepção Visual:** O espectro decompõe a imagem em ondas. A **fase** carrega a inteligibilidade da forma e os contornos; a **magnitude** dita apenas a distribuição de contraste.
- **Eficiência Algorítmica:** O Teorema da Convolução permite que filtros de grande escala sejam aplicados via FFT, despencando a complexidade de $O(N^2K^2)$ para $O(N^2 \log N)$.
- **O Artefato do *Ringing*:** Cortes abruptos de frequência (Filtro Ideal) geram ondulações indesejadas no espaço. Filtros **Gaussianos** e **Butterworth** garantem transições suaves.
- **A Era das Wavelets:** Superando a análise global de Fourier, as Wavelets capturam frequência e localização espacial simultaneamente (multirresolução), fundamentando o JPEG 2000 e a arquitetura das CNNs.

- **Compressão DCT (JPEG):** O algoritmo explora a cegueira humana a altas frequências espaciais. A DCT compacta a energia visual em blocos de 8×8, quantizando e descartando detalhes finos (o que gera artefatos de bloco em qualidades baixas).

  Próximos Passos: O **Capítulo 6** inicia a Parte II da obra, estendendo estas ferramentas aos **Espaços de Cor** e inaugurando os conceitos de **Visão Computacional**.

5.11 Uso do NotebookLM como Tutor Complementar

Nesta edição, incentiva-se o uso do **NotebookLM** como ferramenta complementar de aprendizagem. Baseado em inteligência artificial, o sistema utiliza exclusivamente os documentos fornecidos pelo autor como fonte de conhecimento, produzindo respostas alinhadas ao conteúdo e à abordagem adotada ao longo do livro.

Estude com o Tutor Inteligente

ACESSAR NOTEBOOKLM: CAPÍTULO 05

5.12 Lista de Exercícios

1. **(10%)** Implemente a DFT 2D manualmente (sem `np.fft.fft2`) usando a definição da Equation 5.1 para uma imagem 16×16. Compare com `np.fft.fft2` e verifique que as diferenças absolutas são menores que 10^{-8}. Meça o tempo de execução de ambas as implementações e explique a diferença.

2. **(15%)** Adicione ruído senoidal com frequências $(u_0, v_0) \in \{(5, 10), (20, 5), (30, 30)\}$ à imagem de moedas. Para cada frequência, projete um filtro notch e remova o ruído. Avalie quantitativamente a restauração com PSNR e SSIM. Discuta o compromisso entre remoção de ruído e preservação de detalhes.

3. **(15%)** Compare os filtros passa-baixa Ideal, Gaussiano e Butterworth (ordens $n = 1, 2, 4$) com $D_0 = 20, 40, 60$ pixels. Para cada combinação, calcule o PSNR e o SSIM da imagem filtrada em relação à original. Apresente os resultados em uma tabela e plote as curvas de resposta em frequência $H(u, 0)$.

4. **(15%)** Implemente a decomposição wavelet 2D manualmente usando a wavelet Haar: calcule os filtros h (passa-baixa) e g (passa-alta) de Haar, aplique-os por separabilidade em linhas e colunas, e subamostreie por 2. Compare o resultado com `pywt.dwt2(img, 'haar')` e verifique a equivalência numérica.

5. **(15%)** Aplique o limiamento de coeficientes wavelet (*wavelet thresholding*) com limiares $T \in \{5, 10, 20, 40, 80\}$ e wavelets Haar, db4 e sym4. Para cada combinação, calcule PSNR e SSIM após reconstrução com `pywt.waverec2`. Identifique a combinação wavelet $\times$ limiar que maximiza o SSIM.

6. **(15%)** Implemente o pipeline JPEG completo incluindo: conversão RGB $\rightarrow$ YCbCr, subamostragem 4:2:0 de Cb/Cr, DCT em blocos 8×8, quantização com tabela padrão escalada por fator de qualidade, e reconstrução com IDCT. Compare os resultados com `cv2.imencode('.jpg', img, [cv2.IMWRITE_JPEG_QUALITY, q])` para qualidades 20, 50 e 80.

7. **(15%)** Crie uma imagem sintética composta por três regiões: fotografia de textura natural (canto superior esquerdo), região de texto nítido (centro) e região de gradiente suave (canto inferior direito). Compare JPEG, PNG e WebP para essa imagem mista em termos de PSNR, SSIM e tamanho de arquivo. Justifique qual formato é mais adequado considerando o conteúdo heterogêneo.

Referências do Capítulo

A fundamentação teórica deste capítulo baseia-se nas seguintes obras:

- Gonzalez; Woods (2018) para os conceitos de DFT 2D, filtros no domínio da frequência, DCT e fundamentos de compressão de imagens.

- Oppenheim (1999) para a teoria de sinais e sistemas no domínio discreto, incluindo o Teorema da Convolução e propriedades da DFT.
- Mallat (1999) para a teoria de wavelets, análise multirresolução e bancos de filtros.
- Wallace (1991) para o padrão de compressão JPEG original e o fundamento da quantização DCT.
- Szeliski (2022) para métricas de qualidade (PSNR, SSIM) e comparação de formatos de imagem modernos.

Referências

BARRERA, Junior *et al.* MMach: a mathematical morphology toolbox for the KHOROS system. **Journal of Electronic Imaging**, v. 7, n. 1, p. 174–210, 1998.

BRADSKI, Gary; KAEHLER, Adrian. **Learning OpenCV: Computer vision with the OpenCV library**. *[S.l.]*: "O'Reilly Media, Inc.", 2008.

DOUGHERTY, Edward R.; LOTUFO, Roberto A. **Hands-on morphological image processing**. *[S.l.]*: SPIE press, 2003. v. 59

GONZALEZ, R. C.; WOODS, R. E. **Digital Image Processing**. 4th. ed. New York: Pearson, 2018.

ITU-R. **Recommendation BT.601: Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios**. https://www.itu.int/rec/R-REC-BT.601/, 2011.

KNUTH, Donald E. Literate Programming. **The Computer Journal**, v. 27, n. 2, p. 97–111, 1984.

LEWIS, Patrick *et al.* Retrieval-augmented generation for knowledge-intensive nlp tasks. **Advances in neural information processing systems**, v. 33, p. 9459–9474, 2020.

LOTUFO, Roberto A. *et al.* MMachLib functions and MMach operators. *In*: 1997.

MALLAT, Stéphane. **A wavelet tour of signal processing**. *[S.l.]*: Elsevier, 1999.

MATHERON, Georges. **Random Sets and Integral Geometry**. New York: John Wiley & Sons, 1975.

OPPENHEIM, Alan V. **Discrete-time signal processing**. *[S.l.]*: Pearson Education India, 1999.

OTSU, Nobuyuki. A Threshold Selection Method from Gray-Level Histograms. **IEEE Transactions on Systems, Man, and Cybernetics**, v. 9, n. 1, p. 62–66, 1979.

REDMON, Joseph *et al.* You Only Look Once: Unified, Real-Time Object Detection. *In*: 2016.

SERRA, Jean. **Image Analysis and Mathematical Morphology**. London: Academic Press, 1982. v. 1

SINGH, Himanshu. **Practical machine learning and image processing**. *[S.l.]*: Springer, 2019.

SZELISKI, Richard. **Computer Vision: Algorithms and Applications**. 2. ed. *[S.l.]*: Springer, 2022.

WALLACE, Gregory K. The JPEG Still Picture Compression Standard. **Communications of the ACM**, v. 34, n. 4, p. 30–44, 1991.

ZAMPIROLLI, Francisco de Assis *et al.* A Practical Digital Image Processing Course with morph.py. *In*: Porto Alegre, RS, Brasil: Sociedade Brasileira de Computação (SBC), 2024. Disponível em: <https://sol.sbc.org.br/index.php/educomp/article/view/28199>

ZAMPIROLLI, Francisco de Assis *et al.* Teaching Hands-On Digital Image Processing with morph.py:

Methods and Comprehensive Results. **Revista Brasileira de Informática na Educação**, v. 33, p. 990–1026, 2025.