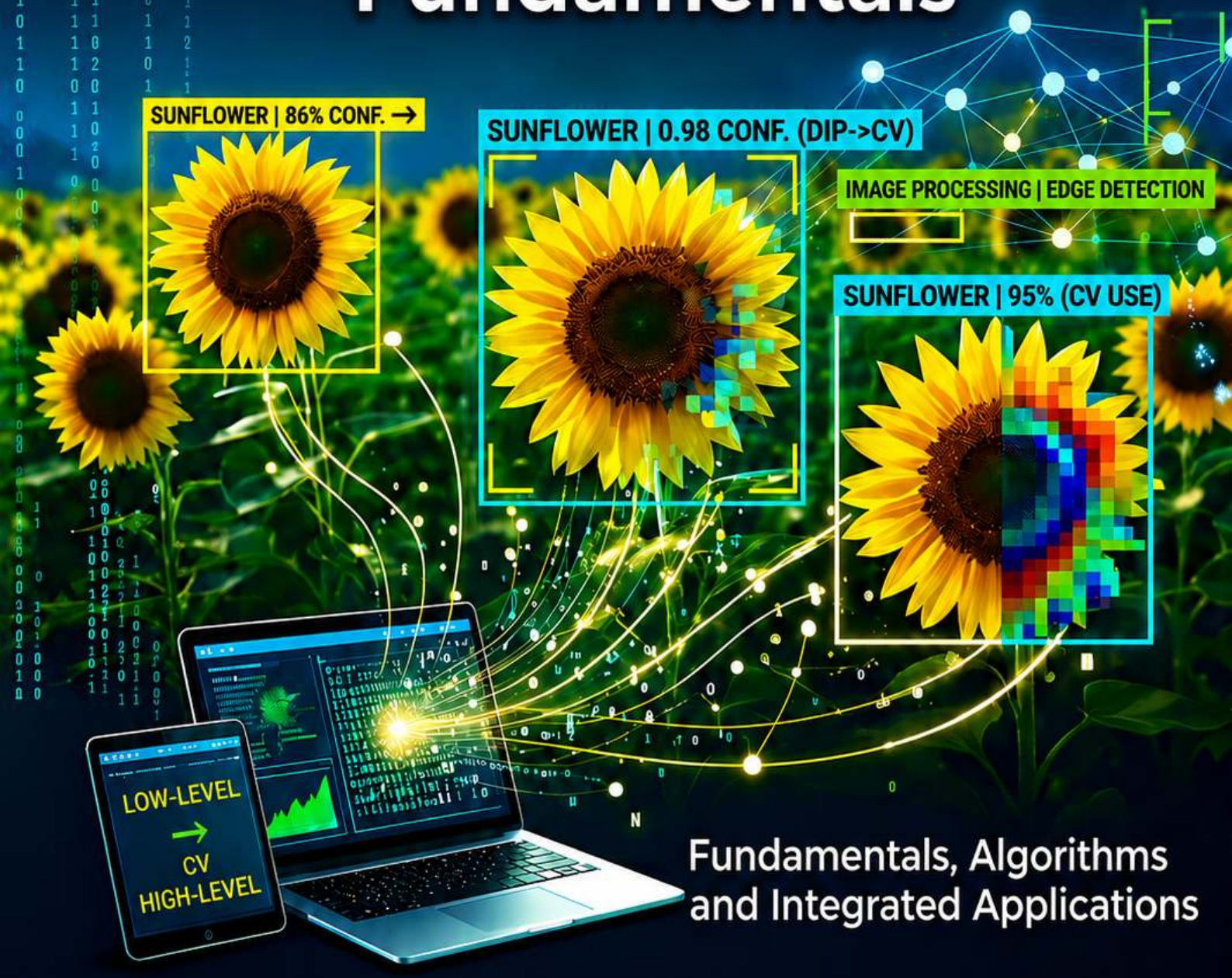
DIP & CV

Digital Image Processing and Computer Vision Fundamentals



Fundamentals, Algorithms and Integrated Applications

AUTHOR: FRANCISCO DE ASSIS ZAMPIROLLI

Digital Image Processing and Computer Vision

Interactive book with Python

Francisco de Assis Zampirolli

Universidade Federal do ABC

September 20, 2026

© 2026 Francisco de Assis Zampirolli, Federal University of ABC (UFABC).
All rights reserved.

This book is licensed under:
Creative Commons Attribution-ShareAlike 4.0 International License
Details at: creativecommons.org/licenses/by-sa/4.0

Graphic design: Francisco de Assis Zampirolli
Cover: Francisco de Assis Zampirolli, with AI support (Gemini and ChatGPT)

CATALOGING-IN-PUBLICATION DATA
UFABC LIBRARY SYSTEM

[CUTTER CODE] Zampirolli, Francisco de Assis
Digital Image Processing and Computer Vision / Francisco de Assis Zampirolli – Santo André, SP : Author's Edition, 2026.

[xx], [NNN] p. : ill.

ISBN: [ISBN TO BE DEFINED] (1st Edition)

1. Digital Image Processing. 2. Computer Vision. 3. Mathematical Morphology. 4. Python. 5. Automatic Grading. 6. Moodle. I. Title.

CDD [XX] ed. – [CDD CODE]

Contents

DIP and CV — Python 1
Organization . . . 1

Preface 3
Context of the first edition . . . 3
How this book is produced . . . 4
Use of Artificial Intelligence tools . . . 5
The `morph.py` library . . . 6
Programming Exercises (EPs) and Automatic Grading . . . 7
Open source . . . 7
Before you begin: Python *Notebooks* . . . 7
Instructor's Guide: Publishing EPs in Moodle . . . 8
Integration with Moodle (VPL) . . . 8
How to Publish to Moodle . . . 9

I Part I — PDI Fundamentals 1

1 Fundamentals and First Steps 3
1.1 Objectives . . . 3
1.2 Before you begin: Interactive *Notebooks* . . . 3
1.3 Fundamentals . . . 3
1.3.1 👁️ Computer Vision . . . 4
1.3.2 🖼️ Digital Image Processing (DIP) . . . 4
1.4 PDI Stages . . . 5
1.5 Image Formation and the Spectrum . . . 5
1.6 What is a Digital Image? . . . 8
Mathematical Representation . . . 8
1.7 Environment Setup . . . 8
1.8 Versions of `morph.py` . . . 9
1.9 Matrix Fundamentals — Caution with Reference Copying . . . 10
1.10 Reading and Displaying Images . . . 11
1.10.1 Alternative: *Downloading* the Image for Local Storage . . . 12
1.11 Type Conversion and Thresholding . . . 13
Conversion to Grayscale . . . 13
Thresholding . . . 13
Practical example of conversion and thresholding . . . 13
1.12 Otsu's Method Thresholding . . . 14
1.13 Accessing Pixels . . . 15
1.14 Summary . . . 17
1.15 🤖 Using Gemini Notebook as a Complementary Tutor . . . 17
Available Platform Features . . . 18
1.16 Exercise List . . . 18
1.17 Chapter References . . . 18
1.18 💻 **Practical Part with Programming Exercises** . . . 19
1.18.1 🎯 Objective of this Notebook . . . 19

1.18.2 § Step-by-Step Instructions . . . 19
1.18.3 ⚠️ Important: Rules and Best Practices . . . 20
1.18.4 EP01_01 📏 Three distance metrics in DIP . . . 21
1.18.5 EP01_02 📊 Predictive Performance — ML Metrics in CV . . . 28
1.18.6 EP01_03 ↗ *Mean Average Precision* (mAP) — Precision-Recall Curve . . . 30
1.18.7 EP01_04 🖼️ Reading and Information from a Matrix Image . . . 34
1.18.8 EP01_05 🔄 Negative of a Grayscale Image . . . 35
1.18.9 EP01_06 🎨 RGB → Grayscale Conversion (ITU-R BT.601) . . . 36
1.18.10 EP01_07 ⚫ Manual Thresholding: Binary Image . . . 38
1.18.11 EP01_08 🎨 Intensity Range Remapping . . . 39
1.18.12 EP01_09 🏁 Checkerboard Pattern: Chess Matrix . . . 41
1.18.13 EP01_10 📄 Metadata: Reading a PGM File . . . 42
1.18.14 EP01_11 ↗ Neighborhood Analysis: 1D Maximum Filter . . . 45

2 From Capture to Pixel - Sampling, Quantization, and Connectivity 47
2.1 Objectives . . . 47
2.2 The Eye and the Camera - Elements of Visual Perception . . . 47
2.2.1 Human vision . . . 47
2.2.2 Digital sensors . . . 47
2.3 Optical Illusions: The Challenges of Visual Perception . . . 48
2.3.1 Ambiguity and Context . . . 48
2.3.2 Brightness and Local Contrast . . . 49
2.3.3 Geometry and Perspective . . . 49
2.4 The Mathematical Model of Image Formation . . . 49
2.4.1 Digital Representation and Spectral Channels . . . 50
2.4.2 Preparing the Practical Environment . . . 51
2.4.3 Image Reading Implementation in `morph.py` . . . 51
2.4.4 RGB and RGBA: Practical Example with the Mandrill Image . . . 52
2.5 Digitization: Sampling and Quantization . . . 53
2.5.1 Sampling - Discretization of Space . . . 53
2.5.2 Quantization - Discretization of Intensity . . . 53
2.5.3 Effects of Sampling and Quantization . . . 53
2.5.4 Technical Analysis . . . 56
2.6 Relationships Between Pixels - Image Topology . . . 56
2.6.1 Neighborhood . . . 57
2.6.2 Adjacency, connectivity, and paths . . . 57
2.6.3 Distances Between Pixels . . . 57
2.7 Image Storage . . . 58
2.7.1 Example: Metadata Extraction and GPS Location . . . 59
2.7.2 Why Is This Separation Important? . . . 60
2.8 Basic Geometric Transformations . . . 61
2.8.1 Translation . . . 61
2.8.2 Rotation . . . 62
2.8.3 Scaling (Resizing) . . . 63
2.8.4 Shearing . . . 64
2.9 Summary . . . 65
2.10 🤖 Using Gemini Notebook as a Complementary Tutor . . . 65
2.11 Exercise List . . . 66
Chapter References . . . 66
2.12 💻 **Practical Part with Programming Exercises** . . . 66
🎯 Objective of this Notebook . . . 67
2.12.1 EP02_01 ⊙ Brightness and Contrast Adjustment . . . 67
2.12.2 EP02_02 🔬 Spatial Subsampling . . . 69
2.12.3 EP02_03 🎨 Gray Level Quantization . . . 70

2.12.4 EP02_04 📐 Distance Transform in Binary Images . . . 72
2.12.5 EP02_05 Image Translation . . . 74
2.12.6 EP02_06 🔄 Image Rotation . . . 77
2.12.7 EP02_07 🔍 Resizing (Scaling) . . . 79
2.12.8 EP02_08 🔀 Shear Transformation . . . 81
2.12.9 EP02_09 🧩 Generic Affine Transformation . . . 82
2.12.10 EP02_10 🎯 Perspective Correction (Homography) . . . 84
2.12.11 EP02_11 🏆 Perspective Correction (Homography) in a Real Image . . . 88

3 Spatial Operations: Intensity, Histogram, and Filtering 93
3.1 Objectives . . . 93
3.2 Intensity Level Operations . . . 93
3.2.1 Preparing the Practical Environment . . . 94
3.2.2 Arithmetic Operations . . . 95
3.2.3 Weighted Mixture (*Alpha Blending*) . . . 95
3.2.4 Logical Operations and Bitwise Masks . . . 97
3.3 Image Histogram . . . 98
3.3.1 Histogram Equalization . . . 99
3.3.2 Histogram Specification . . . 102
3.4 Spatial Fundamentals: Neighborhood, Convolution, and *Kernels* . . . 104
3.4.1 Neighborhood . . . 104
3.4.2 Edge Handling . . . 104
3.4.3 Correlation vs. Convolution . . . 106
3.4.4 The Role of the *Kernel* . . . 107
3.4.5 Numerical Example: Step-by-Step Correlation . . . 110
3.5 Smoothing Spatial Filtering . . . 111
3.5.1 Mean Filter (*Box Filter*) . . . 111
3.5.2 Gaussian Filter . . . 112
3.6 Enhancement Spatial Filtering . . . 114
3.6.1 Laplacian . . . 114
3.6.2 Sobel Operator . . . 118
3.6.3 Prewitt Operator . . . 120
3.6.4 *Unsharp Masking* (USM) . . . 121
3.6.5 Canny Detector . . . 123
3.7 Order Filters: Median Filter . . . 124
3.7.1 Impulsive Noise: Salt and Pepper . . . 124
3.7.2 Median Filter . . . 125
3.8 Practical Application: Preprocessing for Segmentation . . . 128
3.9 Summary . . . 129
3.10 🤖 Using Gemini Notebook as a Complementary Tutor . . . 129
3.11 Exercise List . . . 130
Chapter References . . . 131
3.12 Exercises Proposed (EPs) . . . 131
3.13 Homework: Implement and Test the Following Functions . . . 131
3.14 💻 **Practical Part with Programming Exercises** . . . 131
🎯 Objective of this Notebook . . . 131
3.14.1 EP03_01 ➕ Saturated Addition of a Constant . . . 132
3.14.2 EP03_02 🔀 Alpha Blending of Two Images . . . 133
3.14.3 EP03_03 🎭 Image Inversion (Photographic Negative) . . . 135
3.14.4 EP03_04 📊 Histogram Equalization (L bits) . . . 137
3.14.5 EP03_05 🔲 Binary AND Mask Application . . . 139
3.14.6 EP03_06 N×N Mean Filter *Kernel* . . . 140
3.14.7 EP03_07 🔍 Laplacian Operator (w4) for Edge Enhancement . . . 143
3.14.8 EP03_08 🧭 Sobel Gradient: Gx and Gy . . . 145

3.14.9 EP03_09 📡 3×3 Median Filter . . . 146
3.14.10 EP03_10 ✨ *Unsharp Masking* (USM) . . . 148

4 Mathematical Morphology and Image Segmentation 153
4.1 Objectives . . . 153
4.2 Thresholding . . . 154
4.2.1 Image of Coins . . . 155
4.2.2 Preprocessing for Otsu . . . 155
4.2.3 Result: CLAHE as the Best Pre-processing . . . 158
4.3 Mathematical Morphology . . . 159
4.3.1 Erosion and Dilation . . . 159
4.3.2 Opening and Closing . . . 166
4.3.3 Geodesic Operators . . . 168
4.3.4 Morphological Reconstruction . . . 173
4.3.5 Hole Filling and Border Removal . . . 175
4.3.6 Binary Cleaning Pipeline with CLAHE . . . 177
4.3.7 Grayscale Morphology . . . 179
4.4 Image Segmentation: Fundamentals and Taxonomy . . . 182
4.4.1 Labeling . . . 183
4.4.2 Distance Transform . . . 186
4.4.3 Euclidean Distance Transform in four steps . . . 190
4.4.4 Segmentation by *Watershed* . . . 192
4.5 Shape Component Extraction and Descriptors . . . 197
4.5.1 Labeling and Component Statistics . . . 198
4.5.2 Shape Descriptors . . . 199
4.5.3 Connection with Modern Object Detection . . . 201
4.6 Summary . . . 203
4.7 🤖 Using Gemini Notebook as a Complementary Tutor . . . 204
4.8 Exercise List . . . 204
Chapter References . . . 205
4.9 💻 **Practical Part with Programming Exercises** . . . 205
🎯 Objective of this Notebook . . . 206
4.9.1 EP04_01 Global Thresholding with a Fixed Threshold . . . 206
4.9.2 EP04_02 📊 Otsu's Automatic Thresholding . . . 208
4.9.3 EP04_03 🌱 Flat Binary Dilation (mm.dil0) . . . 210
4.9.4 EP04_04 🪨 Binary Erosion by Flat Structuring Element (mm.ero0) . . . 213
4.9.5 EP04_05 🧹 Morphological Opening (Noise Removal) . . . 214
4.9.6 EP04_06 🧩 Morphological Closing (Filling of Gaps) . . . 218
4.9.7 EP04_07 Weighted Dilation and Erosion (mm.dil1 / mm.ero1) . . . 220
4.9.8 EP04_08 🌋 Morphological Gradient, Top-hat, and Black-hat . . . 221
4.9.9 EP04_09 Distance Transform and the Object's "Core" . . . 223
4.9.10 EP04_10 🪙 *Blob* Separation, Labeling, and Descriptors . . . 225

5 Transforms and Compression 229
5.1 Objectives . . . 229
5.2 Environment Setup . . . 230
5.3 2D Discrete Fourier Transform . . . 230
5.3.1 Simulator: Reconstructing Signals with Sinusoids . . . 230
5.3.2 Interpretation of the Frequency Spectrum . . . 232
5.3.3 The Grid Experiment: Building an Image from a Single Coefficient . . . 232
5.3.4 Mathematical Definition . . . 234
5.3.5 Magnitude and Phase . . . 234
5.3.6 What Do Magnitude and Phase Carry? . . . 235
5.4 Convolution Theorem and Filtering Strategies . . . 237

5.4.1 Separable vs. Non-Separable *Kernel* . . . 237
5.4.2 Computational Efficiency Analysis . . . 238
5.4.3 Discussion of experimental results . . . 238
5.5 Frequency Domain Filters . . . 241
5.5.1 Low-Pass Filters . . . 242
5.5.2 High-Pass and Band-Pass Filters . . . 243
5.5.3 Periodic Noise Removal . . . 247
📌 Synthesis — Spectral Filters . . . 248
5.6 *Wavelets* and Multiresolution . . . 249
5.6.1 The Limit of the Fourier Transform: Spatial Localization . . . 249
5.6.2 2D Discrete Wavelet Transform . . . 250
5.6.3 Wavelet Families . . . 250
5.6.4 Multiresolution Analysis with the 2D DWT . . . 253
Synthesis — Fourier vs. *Wavelets*: when to use each approach? . . . 259
5.7 Image Compression . . . 260
5.7.1 Taxonomy of Redundancies . . . 260
5.7.2 Discrete Cosine Transform (2D DCT-II) . . . 261
5.7.3 The Basis Functions of the DCT . . . 261
5.7.4 Energy Concentration and Progressive Reconstruction . . . 262
5.7.5 The JPEG Compression Pipeline . . . 264
5.7.6 Interactive Simulator: DCT Quantization . . . 267
5.8 Comparison of Image Formats . . . 267
5.8.1 Characteristics of the Formats . . . 268
5.8.2 Quality Assessment Metrics . . . 268
5.8.3 Visual Inspection: Nature of Compression Artifacts . . . 269
5.8.4 Quantitative and Spatial Assessment of Compression . . . 270
Synthesis — JPEG Compression . . . 275
5.9 Practical Application: Noise Removal via Hybrid Filtering . . . 276
5.9.1 Performance Analysis and Chapter Conclusion . . . 276
5.10 Chapter Summary . . . 278
Essential Fundamentals . . . 278
5.11 🤖 Using Gemini Notebook as a Complementary Tutor . . . 280
5.12 Exercise List . . . 280
Chapter References . . . 281
5.13 Chapter 5 - Exercises Proposed (EPs) . . . 281
5.14 Instructions . . . 282
5.15 Exercise 1 . . . 282
5.16 Exercise 2 . . . 282
5.17 Exercise 3 . . . 282
5.18 Exercise 4 . . . 282
5.19 Exercise 5 . . . 282
5.20 Submission . . . 283
5.21 💻 **Practical Part with Programming Exercises** . . . 283
🎯 Objective of this Notebook . . . 283
5.21.1 EP05_01 🟢 Ideal Low-Pass Filter by Distance in the Spectrum . . . 284
5.21.2 EP05_02 🟡 *Notch* Filter: Removing Periodic Peaks . . . 285
5.21.3 EP05_03 🟠 DCT Quantization: The True Source of Compression . . . 287
5.21.4 EP05_04 🔴 Implementing the 2D DFT from the Definition . . . 290
5.21.5 EP05_05 🏆 Complete JPEG Pipeline: DCT, Quantization, and Reconstruction . . . 292

II Part II — Computer Vision **295**

6 Document Analysis and Industrial Inspection **297**

6.1 Chapter Objectives . . . 297
6.2 Environment Setup . . . 298
6.3 Image Databases for Experimentation . . . 299
6.3.1 Public Images with `skimage.data` . . . 300
6.4 Background Normalization and Local Contrast Equalization . . . 302
6.5 Optical Character Recognition (OCR) . . . 304
6.6 Automatic Translation of Recognized Text . . . 306
6.7 Fundamentals of OMR and Industrial Inspection . . . 308
6.8 Practical Projects: Building a Document Analysis *Pipeline* . . . 308
6.8.1 Automatic Document Alignment (*OCR/OMR Pre-processing*) . . . 308
6.8.2 Deskew Algorithm . . . 310
6.8.3 Mathematical Modeling . . . 310
6.8.4 Practical Limitations . . . 313
6.8.5 Edge and Contour Detection . . . 314
6.8.6 Isolation, Segmentation, and Decoding of the *QRCode* . . . 316
6.8.7 Barcode Decoding . . . 321
6.9 The MCTest as a Case Study: From Prototype to Production System . . . 322
6.9.1 Obtaining and Preparing the Module . . . 323
6.9.2 Extraction of the Answer Area . . . 323
6.9.3 *QRCode* Segmentation . . . 326
6.9.4 Location of Answer Boxes . . . 328
6.9.5 Automatic Reading of Answers . . . 328
6.10 Automated Industrial Inspection . . . 330
6.10.1 References and Public Datasets . . . 331
6.10.2 Defect Detection by Texture Analysis . . . 332
6.11 Summary . . . 335
Next Steps . . . 335
6.12 🤖 Using Gemini Notebook as a Complementary Tutor . . . 335
6.13 Exercise List . . . 336
Chapter References . . . 337
6.14 💻 Practical Section with Programming Exercises . . . 337
Difficulty Legend . . . 338
🎯 Objective of this Notebook . . . 338
6.14.1 EP06_01 🟢 Segmentation Evaluation by IoU (*Intersection over Union*) . . . 339
6.14.2 EP06_02 🟢 Marker Filtering by Circularity . . . 341
6.14.3 EP06_03 🟡 Classification of Markings on Answer Sheets (OMR) . . . 343
6.14.4 EP06_04 🟡 Median Angular Slope Estimator (*Deskew*) . . . 345
6.14.5 EP06_05 🟠 Background Normalization by Division (Illumination Correction) . . . 347
6.14.6 EP06_06 🔴 Local Variance Map for Texture Detection . . . 349
6.14.7 EP06_07 🟣 Industrial Inspection Pipeline: Registration by Translation and Subtraction 351
6.14.8 EP06_08 ⚫ QRCode Segmentation and Real Decoding with OpenCV . . . 353

7 Image Classification and Pattern Recognition 359
7.1 Chapter Objectives . . . 359
7.2 Environment Setup . . . 359
7.3 A Concrete Problem: Fruit Classification . . . 361
7.4 Chapter Overview: The Classic Image Classification *Pipeline* . . . 362
7.5 Fundamentals of Pattern Recognition . . . 363
7.6 Extraction of Classical Descriptors . . . 364
7.6.1 *Local Binary Patterns* (LBP) . . . 365
7.6.2 *Histogram of Oriented Gradients* (HOG) . . . 367
7.6.3 The Impact of Scale and Feature Normalization . . . 368
7.7 📌 Concept Map . . . 371
7.8 Descriptors in Practice . . . 371

7.9 A Concrete Problem: Simulating Fruit Descriptors . . . 376
7.10 k-NN Classifier: How It Works Internally . . . 378
7.10.1 Distance Metric . . . 379
7.10.2 Decision Rule . . . 379
7.10.3 The Role of the Parameter k . . . 382
7.11 Practical Project 1: Handwritten Digit Classification with k-NN . . . 384
7.11.1 Classification with Intensity Vectors . . . 384
7.11.2 Classification with HOG Descriptors . . . 386
7.12 Classifier Evaluation . . . 387
7.12.1 Choosing k by Cross-Validation . . . 388
7.12.2 The Bias-Variance Trade-off: Diagnosing *Overfitting* and *Underfitting* . . . 389
7.13 Practical Project 2: Comparison of Descriptors for Texture Classification . . . 391
7.13.1 Feature Extraction Pipeline and Comparative Evaluation . . . 392
7.13.2 Fine-Grained Classifier Diagnostics: Precision, Recall, and F1-Score . . . 395
7.14 Limitations of Handcrafted Descriptors . . . 397
7.15 Summary . . . 397
7.16 🤖 Using Gemini Notebook as a Tutor . . . 398
7.17 Exercise List . . . 398
Chapter References . . . 399
7.18 💻 **Practical Part with Programming Exercises** . . . 400
🎯 Objective of this Notebook . . . 400
Download . . . 400
Summary of the `morph.py` Methods (Ch. 7) . . . 401
7.18.1 EP07_01 🟢 Step-by-Step k-NN Classifier . . . 401
7.18.2 EP07_02 🟡 *Z-score* Normalization and Robustness of k-NN to Distinct Scales . . . 403
7.18.3 EP07_03 🟡 Evaluation via Confusion Matrix . . . 405
7.18.4 EP07_04 🟠 Manual Encoding of the LBP Descriptor . . . 407
7.18.5 EP07_05 🔴 Histogram of Orientations for an HOG Cell . . . 409
7.18.6 EP07_06 🟣 Complete *Pipeline*: Descriptors + k-NN + Multi-Class Evaluation . . . 411
7.18.7 EP07_07 ⚫ Real Classification of a Texture Mosaic via LBP + k-NN . . . 414

8 Understanding Scenes: Feature Matching, Detection, and Segmentation 423
8.1 Chapter Objectives . . . 423
8.2 Environment Setup . . . 423
8.3 Local Feature Detectors and Descriptors . . . 425
8.3.1 The ORB Algorithm (*Oriented FAST and Rotated BRIEF*) . . . 425
8.3.2 Exploring the ORB Simulator . . . 426
8.4 Feature Matching and Homography with ORB . . . 427
8.4.1 Detecting and Matching Features with ORB . . . 429
8.5 Mathematical Modeling: Homography and RANSAC . . . 431
8.5.1 Homography . . . 431
8.5.2 RANSAC . . . 431
8.5.3 Estimating Homography with RANSAC . . . 433
8.5.4 Registering the Image . . . 434
8.6 Object Detection: *Haar Cascade* (Viola-Jones) . . . 436
8.7 Object Detection: Bounding Boxes, IoU, and NMS . . . 438
8.7.1 *Intersection over Union* (IoU) . . . 438
8.7.2 Non-Maximum Suppression (NMS) . . . 439
8.8 Visual Segmentation: Semantic, Instance, and Panoptic . . . 440
8.8.1 Semantic Segmentation by Thresholding . . . 441
8.8.2 Instance Segmentation by Labeling . . . 441
8.8.3 Alternative: Distance Transform + *Watershed* . . . 443
8.8.4 Measurement and Extraction of Attributes with `mm.measure` . . . 444
8.8.5 Exporting Annotations: from `mm.measure` to the YOLO Format . . . 446

8.8.6 Validation of Annotations by IoU: `mm.verifyBoundBox` . . . 447
8.8.7 Viewing Saved Annotations: `mm.showBoundBox` . . . 448
8.9 Overview of Modern Models . . . 449
8.10 Limitations of Classical Approaches and Motivation for *Deep Learning* . . . 450
8.11 Summary . . . 450
Next Steps . . . 451
8.12 🤖 Using Gemini Notebook as a Study Aid . . . 451
8.13 Exercise List . . . 451
Chapter References . . . 452
8.14 💻 **Practical Part with Programming Exercises** . . . 453
🎯 Objective of this Notebook . . . 453
Download . . . 453
Summary of Methods in `morph.py` (Ch. 8) . . . 453
8.14.1 EP08_01 🟢 Hamming Distance and Binary Descriptor Matching . . . 454
8.14.2 EP08_02 🟢 Homography and RANSAC: Voting by *Inliers* . . . 456
8.14.3 EP08_03 🟢 Integral Image: Rectangular Sums in Constant Time . . . 458
8.14.4 EP08_04 🟢 IoU and Non-Maximum Suppression (NMS) . . . 460
8.14.5 EP08_05 🟡 Connected Component Labeling: Instance Segmentation . . . 462
8.14.6 EP08_06 🟡 *Bounding Boxes*, Centroids, and Instance Properties with `mm.measure` . 464
8.14.7 EP08_07 🟡 Salt-and-Pepper Noise Removal and Object Measurement . . . 466
8.14.8 EP08_08 🟡 Grayscale Image and Dynamic Thresholding . . . 468
8.14.9 EP08_09 🟠 Illumination Gradient and Adaptive Thresholding . . . 470
8.14.10 EP08_10 🔴 Low Contrast and Object Separation Topics (*Watershed* / Distance) . . 472
8.14.11 EP08_11 🔴 Classification and Validation of Objects with Bounding Box Ground Truth 473
8.14.12 EP08_12 🔴 Instance Segmentation on Real Images: Geometric Objects . . . 476

9 Deep Learning for Computer Vision **481**
9.1 Chapter Objectives . . . 481
9.2 Overview: Classification, Detection, and Segmentation . . . 482
9.3 Environment Setup . . . 483
9.4 Fundamentals of Deep Learning for CV . . . 484
9.4.1 From Fixed Convolution to Learned Convolution . . . 485
9.4.2 Artificial Neuron . . . 485
9.4.3 Convolutional Layer . . . 486
9.4.4 Activation Function . . . 488
9.4.5 *Pooling* . . . 488
9.4.6 Training Neural Networks: How CNNs Learn . . . 488
9.4.7 Architecture of a CNN . . . 494
9.4.8 How the Gradient Adjusts Convolution Kernels . . . 494
9.5 Practical Applications in CV . . . 499
9.5.1 Image Classification with CNNs . . . 499
9.5.2 Object Detection . . . 529
9.5.3 Object Segmentation . . . 539
9.5.4 Data Engineering for CV (*Roboflow*) . . . 549
9.5.5 Geometric and Integrated Applications . . . 553
9.6 Summary . . . 558
9.7 🤖 Using Gemini Notebook as a Complementary Tutor . . . 559
9.8 Exercise List . . . 559
9.9 Closing of Part II . . . 561
Chapter References . . . 561
9.10 💻 **Practical Section with Programming Exercises** . . . 562
🎯 Objective of this Notebook . . . 563
9.10.1 EP09_01 🟢 Manual 2D Convolution (Forward Pass of a Learned Layer) . . . 563
9.10.2 EP09_02 🟢 Manual *Pooling* (Maximum and Average) . . . 565

9.10.3 EP09_03 🟡 Counting Trainable Parameters of a CNN . . . 568
9.10.4 EP09_04 🟡 Intersection over Union (IoU) and Non-Maximum Suppression (NMS) . 573
9.10.5 EP09_05 🟠 Segmentation Evaluation: Pixel-wise IoU and Dice . . . 577
9.10.6 EP09_06 🔴 Integrated Pipeline: From Detection to Real-World Measurement . . . 580

References **583**

Appendices **585**

A About MCTest **585**
A.1 Installing MCTest . . . 585
A.2 Creating an exam in MCTest . . . 585
A.3 Creating a parametric question . . . 586
A.4 Real example: a question from Mock Exam 4 . . . 588
A.5 Final remarks . . . 590

B About Moodle (VPL) **591**
B.1 Configuring a VPL activity . . . 591
B.2 Publishing EPs as VPL activities . . . 591
B.3 Final remarks . . . 592

C Using SEB in Biweekly Mock Exams and Assessments **593**
C.1 Configuring the application in *SEB Tool* . . . 593
C.2 Linking the SEB key to the VPL activity in Moodle . . . 594
C.3 Restriction by IP address (optional) . . . 594
C.4 Final remarks . . . 594

D Generating Course Material with *Gemini Notebook* **595**
D.1 Conceptual videos and EP-solving videos . . . 595
D.2 Supporting *slides* . . . 595
D.3 Main material and final remarks . . . 595

E Using AI in Student Assessment: the vpl-ai-feedback Project **597**
E.1 Context of use . . . 597
E.2 Project architecture . . . 598
E.3 The universal *prompt* and the two-stage rubric . . . 599
E.4 Two additional layers of evidence . . . 600
E.5 Execution flow . . . 600
E.6 Illustrative example: rubric for a three-question exam . . . 601
E.7 Sending *feedback* by e-mail . . . 603
E.8 Risks, ethical limits, and instructor responsibility . . . 605
E.9 Final remarks . . . 605

List of Figures

1 Clickable **Run on Colab** button, available at the beginning of the **Theoretical** and **Practical** parts of each chapter, allowing interactive execution of the examples and exercises. In the PDF version, there is a direct HTML *link* between the simulator image and its caption. . . . 5

1.1 Relational diagram detailing the fundamental distinctions, synergies, and interconnections between DIP and CV in the context of image-based systems. . . . 4
1.2 Representation of the sequential processing flow: the output of each level becomes the input of the subsequent level. . . . 6
1.3 Detailed DIP flow: from sensory acquisition to attribute extraction and automated recognition, including color processing and compression. . . . 6
1.4 Representation of the visible spectrum and its position relative to other electromagnetic radiations, highlighting the variation in wavelengths from 380 nm to 750 nm. . . . 7
1.5 (A) Complete electromagnetic spectrum on a logarithmic scale, highlighting the visible range. (B) Detail of visible light (380–750 nm) and its colors. (C) Decomposition of white light by the prism: shorter wavelengths undergo greater refraction, separating UV, visible, and infrared light. 7
1.6 Examples of synthetic images represented as matrices. . . . 11
1.7 Mandrill (Mandrillus sphinx) in natural environment. Credit: Julien Renoult (CC BY 4.0). . 12
1.8 Basic image processing: (a) original image, (b) grayscale image, (c) image binarized by threshold (T=128). . . . 14
1.9 Comparison between manual thresholding (T=128) and automatic (Otsu) on the grayscale image. . . . 15
1.10 Accessing a specific pixel (r, c) and the representation of its neighborhood in the image. . . . 16
1.11 EP01_01 Simulator: Euclidean, City-block, and Chessboard Distances . . . 23
1.12 EP01_02 Simulator: Predictive Performance — ML Metrics . . . 30
1.13 Simulator EP01_03: Mean Average Precision (mAP) and P-R Curve . . . 33
1.14 EP01_04 Simulator: Pixel Statistics in Discrete 5x5 Matrix . . . 35
1.15 EP01_05 Simulator: Negative Image Transformation (Intensity Inversion) . . . 37
1.16 EP01_06 Simulator: RGB to Grayscale Conversion (Perceptual ITU-R BT.601 Weighting) . 38
1.17 Simulator EP01_07: Interactive Global Thresholding (Binarization p > T) . . . 40
1.18 EP01_08 Simulator: Conditional Range Remapping (Brightness and Contrast by Threshold) 41
1.19 EP01_09 Simulator: Checkerboard Pattern Generator (Parity Logic (i + j) % 2) . . . 43
1.20 Simulator EP01_10: PGM File Structure (ASCII P2 Header and Mapping to Python Tuple) 44
1.21 EP01_11 Simulator: 1D Local Maximum Filter (1x3 Neighborhood with Border Condition) . 46

2.1 Didactic comparison between the biological and electronic visual systems: at the top, the anatomy of the human eye highlighting the retina and photoreceptors (cones and rods); below, the structure of a digital camera detailing the CMOS sensor, the Bayer filter array (RGGB), and the digital quantization process carried out by the ADC. . . . 48
2.2 Collection of perceptual challenges: (top left) Rubin Vase — figure-ground ambiguity; (top center) Shepard Elephant — geometric incongruity; (top right) Adelson Shadow — brightness constancy based on context; (bottom left) Dot grid — lateral inhibition; (bottom right) Schröder Staircase. . . . 49
2.3 RGB image (3 channels, $M \times N \times 3$) and RGBA version (4 channels, $M \times N \times 4$) with gradual alpha transparency from left to right. Mandrill image — USC SIPI Image Database (public domain). . . . 53
2.4 Effect of subsampling. The titles display the dimensions (W x H) and the size of the array in memory (KB). . . . 55

2.5 Effect of reducing bit depth. The titles display the number of bits, levels, and size in memory (KB). . . . 56
2.6 Illustration of 4-neighborhoods in a 3x3 matrix. At the center (1,1), the pixel of interest. . . . 57
2.7 Environmental Protection Area Quilombos of Médio Ribeira - Barbudo-rajado (Malacoptila striata). Credit: Thomas Fuhrmann (CC BY-SA 4.0). . . . 60
2.8 Example of translation of the bird image with offsets (50,50) and (100,50). . . . 62
2.9 Example of rotating the bird image by 30° and 45° using bilinear interpolation. . . . 63
2.10 Comparison of interpolation with zoom on the eye detail (60×60 crop, enlarged 4×). Note the blocky effect in nearest neighbor vs. the smoothing in bilinear. . . . 64
2.11 Example of bird image shearing: horizontal (shx=0.3), vertical (shy=0.3) and combined (shx=0.2, shy=0.2). . . . 65
2.12 Simulator EP02_01: Linear Brightness and Contrast Adjustment (p' = p +) . . . 69
2.13 EP02_02 Simulator: Spatial Subsampling (Resolution Reduction by f-Step) . . . 71
2.14 EP02_03 Simulator: Quantization and Bit Depth (Reduction of Gray Levels) . . . 73
2.15 EP02_04 Simulator: Distance Transform in Binary Image (Chessboard, City-block and Euclidean) . . . 75
2.16 EP02_05 Simulator: Geometric Image Translation (Displacement tx and ty with Border Filling) 76
2.17 EP02_06 Simulator: Image Rotation Around Origin by Angle . . . 78
2.18 EP02_07 Simulator: Spatial Resizing and Interpolation (Nearest Neighbor vs Bilinear) . . . 80
2.19 EP02_08 Simulator: 2D Shear Geometric Transformation (Horizontal and Vertical Shear) . . 83
2.20 Simulator EP02_09: Affine Transformation 2D (2×3 Matrix) . . . 85
2.21 Simulator EP02_10: Perspective Correction (3×3 Homography Transformation) . . . 87
2.22 Acquisition of a Sudoku image on the left. On the right, conversion to grayscale and resizing. Credit: Héctor Rodríguez de Guardamar, Spain (CC BY 2.0). . . . 89
2.23 Perspective correction: original and rectified front view. . . . 90
2.24 EP02_11 Simulator: Sudoku Perspective Correction (3×3 Homography with Bilinear Resampling) . . . 91

3.1 *Mandrill* (*Mandrillus sphinx*) photographed in its natural habitat in South Africa. Credit: Carlos Guilherme Rodrigues (CC BY-SA 3.0). . . . 94
3.2 Saturated arithmetic operations: constant addition (lightening) and constant subtraction (darkening with saturation at 0). . . . 95
3.3 Portrait of a leopard (*Panthera pardus*) in its natural habitat. Credit: C. Brück (CC BY-SA 4.0). . . . 96
3.4 *Alpha blending* between aligned crops of mandrill and leopard (Figure 3.3) for different values of . At =1 only the mandrill is seen; at =0, only the leopard; intermediate values merge the gazes of the two images proportionally. . . . 97
3.5 Bitwise logical operations with a circular mask: AND (ROI isolation), OR (ROI illumination) and NOT (negative). . . . 98
3.6 Histograms of the original image, a darkened version (−80), and a brightened one (+80). Subtraction/addition saturates at 0 and 255. . . . 99
3.7 Histogram equalization in a 5×5 image with 3 bits (L=8): original image with tones concentrated in {2,3,4}, equalized image with tones redistributed to {1,5,7}, and the respective histograms showing the spreading of frequencies. . . . 101
3.8 Histogram equalization: mm.equalize (global via CDF) vs. CLAHE (adaptive with contrast limitation). The histograms reveal the progressive spread of intensities. . . . 102
3.9 Histogram specification: mandrill mapped to the leopard tonal profile. The output CDF approximates the reference CDF. . . . 103
3.10 Correlation with 3×3 averaging *kernel*: didactic version (mm.conv0) vs. vectorized (mm.conv via cv2.filter2D). The maximum difference of 39 occurs at the borders. . . . 110
3.11 *Patch* 5×5 extracted from the leopard image (position [250:255, 250:255]). The yellow window highlights the 3×3 neighborhood centered on pixel [1,1] where the correlation will be computed. 110
3.12 Average filter with kernels of increasing size (3×3, 7×7, 15×15). The blurring of the edges increases with the kernel size. . . . 112

3.13 *Gaussian* Kernel 5×5 (=1): normalized weights, larger at the center and decreasing radially. The grid facilitates reading each coefficient. . . . 113
3.14 Comparison between the mean filter and Gaussian (*kernel* 9×9, =0). The Gaussian preserves edges better, visible in the detail of the face. . . . 114
3.15 Simulator: 1D Spatial Enhancement Filtering (Unsharp Masking and High-Boost) . . . 115
3.16 *Laplacian kernel* w4 applied to the 5×5 *patch*: the yellow window highlights the 3×3 neighborhood where the second derivative operator is computed. . . . 117
3.17 Laplacian applied to the leopard image: raw response (edges) with w4 and w8, and images enhanced by subtracting the Laplacian. w8 is more sensitive to diagonals. . . . 118
3.18 Sobel operator on the leopard image: Gx detects vertical edges, Gy detects horizontal edges, and the magnitude | f| combines both, revealing all edges regardless of direction. . . . 120
3.19 Prewitt operator: Gx, Gy and magnitude — comparable to Sobel, but without perpendicular Gaussian weighting. . . . 120
3.20 Sharpening by *Unsharp Masking* on the leopard image: the smoothed image is subtracted from the original to generate the high-frequency mask, which is then reintroduced to enhance edges and details. . . . 122
3.21 *Unsharp Masking* on the leopard image with =1 and k from 0.5 to 8.0. For k>2 halos appear at the edges and the background noise becomes visible. . . . 123
3.22 Canny detector with different threshold pairs: low thresholds capture more edges (including noise); high thresholds retain only the strongest edges. . . . 124
3.23 Salt and pepper noise with increasing densities (2%, 5%, 10%). The parameter prob indicates the total fraction of corrupted pixels, half salt (255) and half pepper (0). . . . 125
3.24 Comparison of filters for removing salt-and-pepper noise (10%): Gaussian, Mean, Median, Bilateral, and Morphological. Top row: full images; bottom row: crop of the region of interest. 128
3.25 *Pipeline* de pré-processamento: CLAHE → Gaussiano → Canny. Cada etapa prepara a imagem para a seguinte, resultando em bordas limpas e bem definidas. . . . 129
3.26 Simulador EP03_01: Saturated Addition of Constant (p' = clip(p + k)) . . . 133
3.27 Simulator EP03_02: Alpha Blending of Two Images (g = · f1 + (1−) · f2) . . . 135
3.28 EP03_03 Simulator: Image Inversion — Photographic Negative (p' = 255 − p) . . . 136
3.29 Simulator EP03_04: Histogram Equalization (Levels L = 2^B) . . . 138
3.30 EP03_05 Simulator: Binary AND Mask Application . . . 140
3.31 Simulator EP03_06: Average Filter with N×N Kernel . . . 142
3.32 Simulator EP03_07: Laplacian Operator (w4) for Edge Enhancement . . . 144
3.33 EP03_08 Simulator: Sobel Gradient (Gx and Gy) . . . 147
3.34 EP03_09 Simulator: 3×3 Median Filter . . . 149
3.35 EP03_10 Simulator: *Unsharp Masking* (USM) . . . 151

4.1 Image with coins of various types. Credit: GAZI.MD.AHAD (CC BY-SA 4.0). . . . 155
4.2 Comparison of preprocessings: image | histogram+T* | ²B(T) | Otsu. The best bimodal separation indicates the most reliable threshold. . . . 158
4.3 Structuring element in the shape of a cross (B_{cross}) used for 4-connectivity. . . . 160
4.4 Erosion and dilation on a 10×10 binary image with an asymmetric 3×3 structuring element 'L'. Validation of the erosion–dilation duality. . . . 166
4.5 Simulator: Morphological Erosion (A B_L) . . . 166
4.6 Opening, closing, composition, and alternating sequential filter applied to Otsu binarization of the coins (preprocessed by contrast enhancement). Structuring element: 13×13 disk. . . . 168
4.7 Interactive simulator of geodesic dilation: the marker f (green) propagates step by step through the free paths of mask g, without crossing walls. . . . 170
4.8 Maze resolution via Complementary Operations: Inverting the mask and marker at the start of the *pipeline*, the flow is resolved directly in the complementary domain through mm.cero and mm.suprec, eliminating redundant re-inversions. . . . 173
4.9 *Morphological Reconstruction Pipeline by Conditioned Dilation: the mask contains two objects, the marker isolates only the nucleus of the main object, and the iterations rebuild its exact shape until convergence. . . . 175

4.10 *Hole-filling pipeline* (*clohole*): the marker comes from the image border, restricted to the complement f^c. Geodesic dilation reconstructs the external background; after complementation, the internal holes are filled. 177
4.11 *Pipeline* for removing border structures (*edgeoff*): the marker captures roots connected to the extremities, reconstruction delimits these elements, and subtraction preserves only fully internal objects. 177
4.12 Complete *pipeline* of segmentation with CLAHE: Otsu → opening (r=9) → clohole → opening (r=33) → edgeoff. 178
4.13 Advanced interactive morphology simulator with editable structuring element and 1D profile. 180
4.14 Grayscale morphology and their respective histograms. Erosion reduces local intensities, dilation amplifies them, the gradient highlights edges, and the Top-hat and Black-hat operators highlight bright and dark local details. Structuring element: disk of diameter 19. 182
4.15 Flood-fill labeling algorithm with stack. 185
4.16 Interactive simulator for connected component labeling: visualization of flood-fill expansion, 4-connectivity and 8-connectivity. 186
4.17 Effect of connectivity on labeling: the 10×10 binary image contains diagonally adjacent pixels. With 4-connectivity, these pixels form distinct components; with 8-connectivity, they merge into a single component. 186
4.18 Distance Transform Algorithm. 188
4.19 Interactive simulator of the Distance Transform (DT) iterative via grayscale erosion. Pixels outside the image assume the maximum value (144), propagating costs from the internal background. 189
4.20 Distance Transform on a 10×10 binary image. Left: original (*foreground* = 255). Center: mm.dist1 iterative (erosions with cross). Right: mm.dist (L2). 189
4.21 Interactive 2D simulator (4x4) with strict synchronization of horizontal propagation queues (b) to obtain the exact convergence described in the paper. 190
4.22 Shortest geodesic path in a maze. Geodesic distances are calculated from the entrance and exit using mm.gdist. Pixels whose sum of the two distances equals the minimum distance between markers belong to an optimal path. 192
4.23 Didactic *Watershed* Algorithm by Region Growing Limited by Mask. 193
4.24 Interactive simulator of the *Watershed* algorithm by morphological propagation. Draw the mask, position the markers and adjust the Structuring Element to observe the flooding. When basins meet simultaneously, the tie is resolved by randomly assuming one of the regions. . . . 194
4.25 *Pipeline watershed* delimited by mask on a 20×20 binary image. 196
4.26 Watershed *Pipeline* for separating overlapping coins: from the dilated binary mask to the final annotated contours on the original image. 197
4.27 Connected components extracted after the CLAHE → Otsu → morphological cleanup *pipeline*. Each object is colored with a distinct color and annotated with its area in pixels. 199
4.28 Contours extracted with *findContours*. Each coin is annotated with its circularity — values close to 1 confirm a circular shape. 201
4.29 *Bounding boxes* derived from the connected components overlaid on the original image. The annotations are exported in YOLO format (*class cx cy w h*), with coordinates normalized to the range [0,1]. 202
4.30 EP04_01 Simulator: Global Thresholding by Fixed Threshold (p' = (p > T) ? 255 : 0) . . . 208
4.31 EP04_02 Simulator: Automatic Otsu Thresholding (T* = argmax ²_B(T)) 210
4.32 Simulator EP04_03: Binary Dilation (g = f B) 212
4.33 EP04_04 Simulator: Planar Binary Erosion (g = f B) 215
4.34 EP04_05 Simulator: Morphological Opening (g = (f B) B) 217
4.35 EP04_06 Simulator: Morphological Closing (g = (f B) B) 219
4.36 EP04_07 Simulator: Dilation and Erosion with Weights (mm.dil1 / mm.ero1) 222
4.37 EP04_08 Simulator: Morphological Gradient, Top-hat and Black-hat 224
4.38 Simulador EP04_09: Distância por Transformada (Camadas de Erosão) 226
4.39 Simulador EP04_10: Blob Separation, Labeling, and Descriptors 228

5.1 1D Fourier decomposition: a square wave (dashed line) is approximated by the sum of the first sinusoids (colored lines). The more terms, the better the approximation. 230
5.2 Interactive simulator of 1D Fourier decomposition: visualization of the sum of sinusoids with different frequencies, amplitudes and phases. Add terms and observe convergence to arbitrary waveforms. 231
5.3 Every frequency in the spectrum (isolated point) corresponds to a rotated 2D sine wave in the spatial domain. 233
5.4 Interactive simulator of 2D Fourier synthesis. Change the horizontal (u) and vertical (v) position of the coefficient in the centered frequency spectrum and observe how the distance from the center dictates the spatial frequency (thickness) and the angle dictates the orientation of the generated sine wave. 233
5.5 Conceptual diagram of the centered 2D Fourier spectrum. 235
5.6 Phase swap experiment: Image A (coins) and Image B (geometric pattern) reconstructed with swapped magnitudes and phases. The result shows that the visual structure is **much more sensitive to phase** than to magnitude: when B's phase is preserved, the resulting image retains B's spatial organization, even with A's magnitude. Magnitude, in turn, mainly influences contrast and texture. Note that reconstruction quality is not perfect — visible artifacts exist —, highlighting the interdependence between phase and magnitude for a faithful image representation. 237
5.7 Efficiency comparison: Non-Separable Convolution (Spatial 2D), Separable (Spatial 1D) and via FFT. 239
5.8 Without *padding*, a severe shift makes the image leak to the opposite side (circular convolution). 240
5.9 Verification of the Convolution Theorem: the pixel-by-pixel difference between spatial convolution (cv2.filter2D) and frequency multiplication (FFT) is numerically zero — confirming the theoretical equivalence. 241
5.10 The Dangerous Duality: The abrupt cutoff in Frequency (Cylinder) necessarily transforms into a spatial Sinc. Its ripples cause the ghostly *ringing* at the edges of the image. 242
5.11 Low-pass filters. 243
5.12 Interactive simulator of filters in the frequency domain. 244
5.13 Comparison between low-pass filters: Ideal (D =30), Gaussian (D =30) and Butterworth (D =30, n=2). H(u,v) profiles along a central line and corresponding filtered images. 246
5.14 Gaussian high-pass filter. (a) Original; (b) High-pass filter (D =30) - the edges of the coins and textured background are enhanced. 247
5.15 Periodic noise removal via *notch* filter in the frequency domain: (a) image with sinusoidal noise, (b) spectrum showing noise peaks, (c) *notch* mask centered on the peaks, (d) restored image. 248
5.16 Fourier global is blind to position. The spectra do not tell where the edges are. 250
5.17 Wavelet functions ( ). Note how they quickly decay to zero (compact support), unlike the infinite sinusoids of Fourier. 252
5.18 Diagram of the 2D wavelet decomposition at two levels. 252
5.19 Simulation of 2D *wavelet* decomposition. 254
5.20 2D *wavelet* decomposition of 2 levels with Haar *wavelet*: LL, LH, HL, HH subbands at each level. The detail subbands reveal oriented structures at different scales using a synthetic pattern. 256
5.21 Comparison between *wavelet* families: Haar, db4, sym4, and bior2.2. LL subband (approximation) and HH (diagonal) for each choice, illustrating the trade-off between compactness and smoothness based on the synthetic pattern. 257
5.22 Wavelet reconstruction with coefficient thresholding (*hard thresholding*): as the threshold increases, more details are zeroed out, producing progressively smoother images. The PSNR metric quantifies the quality loss on the synthetic pattern. 259
5.23 The Visual Alphabet of JPEG: The 64 basis functions of the DCT-II. The DC coefficient is at the top left (smooth). Going down and to the right, the spatial oscillation increases drastically. 262
5.24 2D DCT in 8×8 block: coefficients and progressive reconstruction. 264

5.25 *Pipeline* simplified JPEG applied to the classic *Cameraman* image: block DCT 8×8, quantization with different quality factors and reconstruction via IDCT. Blocking artifacts become visually evident at reduced quality factors ($Q = 10$ and $Q = 25$). 267
5.26 Interactive DCT-JPEG compression simulator: adjust the quality factor and visualize in real time the zeroed coefficients, the reconstructed block, and the quantization error. 268
5.27 Comparative analysis of compression artifacts under reduced quality factor ($Q = 10$). On the left, the block artifact characteristic of DCT discretization in JPEG is observed. On the right, the attenuation and edge smoothing effect intrinsic to the WebP standard is evident. 270
5.28 Rate-distortion curve: PSNR vs file size for JPEG, WebP, and PNG applied to the *Cameraman* image. 272
5.29 Spatial degradation analysis: reconstructed images and respective normalized absolute error maps for different JPEG quality factors. 274
5.30 Full mixed noise removal *pipeline*: (1) addition of Gaussian and periodic noise; (2) identification of interference peaks in the frequency spectrum; (3) application of *notch* mask with smooth Gaussian attenuation; (4) post-processing via bilateral filter to eliminate residual stochastic noise. 278
5.31 Conceptual map of transformations and properties in the frequency domain. 279
5.32 EP05_01 Simulator: Ideal Low-Pass Filter in the Spectrum 286
5.33 EP05_02 Simulator: Notch Filter 288
5.34 EP05_03 Simulator: DCT Quantization (*round-trip*) 289
5.35 EP05_04 Simulator: Manual 2D DFT 291
5.36 EP05_05 Simulator: Complete JPEG pipeline in block 293

6.1 Sample of public images from *skimage.data*, grouped by application area. 302
6.2 Comparison between preprocessing strategies for text image binarization: (a) original image; (b) direct thresholding by Otsu's method; (c) background estimated by Gaussian filtering; (d) image after background normalization (division by the smoothed image); (e) CLAHE followed by Otsu thresholding; and (f) background normalization followed by Otsu thresholding. The images illustrate the effect of each technique on compensating for illumination variations and on segmentation quality. 303
6.3 Comparison of the text extracted by Tesseract from four versions of the same image: (a) original image; (b) CLAHE followed by Otsu thresholding; (c) background normalization followed by Otsu thresholding; and (d) background normalization in grayscale, without thresholding. The comparison shows that external binarization does not always favor recognition, since Tesseract already performs its own adaptive binarization internally. 306
6.4 Automatic recognition and translation flow. The image pre-processed by background normalization, without thresholding, is used as input to Tesseract OCR, and the recognized text is translated from English to Portuguese using the *deep-translator* library. 307
6.5 *Pipeline* for document ingestion: adaptive rasterization of PDF pages into discrete arrays in PNG format, showing page two. 310
6.6 Interactive skew correction simulator (*deskew*): move the *slider* to tilt the document and observe the three steps of the *pipeline* — tilted image, Canny edges, and corrected result. . . 312
6.7 *Axial rectification pipeline*: comparative display between the original rotated input, the Canny structural gradient map, and the final aligned result with normalized white background. . . . 313
6.8 Detection of marker discs, contour extraction, and rectification by perspective transformation. 316
6.9 Interactive simulator of circularity filtering: move the *slider* to adjust the threshold C and observe which components are accepted (green) or rejected (red). 317
6.10 *Pipeline* of *QRCode* processing: thresholding, morphological opening, inversion, and final cropping for decoding. 319
6.11 Interactive simulator of the QRCode isolation *pipeline*: navigate through the filtering steps, adjust the structuring element of the morphological closing, and see the square contour detection on the original template image. 320
6.12 Decoding linear barcodes with *pyzbar*: input image and extracted data. 322
6.13 Image of the grayscale answer sheet loaded from the rasterized PDF. 325

6.14 Answer area extracted by *getAnswerArea*: rectified region containing the marking frames. . . 326
6.15 Region of the *QRCode* isolated by *segmentQRcode* within the rectified response area. 327
6.16 Lower crop of the response area, focusing on the bubble frames to be segmented. 328
6.17 Answers read automatically by MCTest after segmentation and classification of all bubbles. . 330
6.18 Defect detection by image subtraction: reference product, image with simulated defect, and detected anomaly mask. 332
6.19 Interactive industrial inspection simulator by image subtraction: control the geometric misalignment distortions (registration) and the detection threshold to observe the impact on false positives. 333
6.20 Detection of texture heterogeneity: local variance map and anomaly mask. 334
6.21 EP06_01 Simulator: IoU between reference mask and predicted mask 341
6.22 Simulator EP06_02: Marker Filter by Circularity 343
6.23 EP06_03 Simulator: OMR Mark Classification 345
6.24 EP06_04 Simulator: Slope Estimator by Angular Median 347
6.25 EP06_05 Simulator: Background Normalization by Division 348
6.26 EP06_06 Simulator: Local Variance Map for Texture Detection 351
6.27 EP06_07 Simulator: *Pipeline* Inspection — Registration by Translation and Subtraction . . . 353
6.28 Simulador EP06_08: Geometric Segmentation + Verification by QRCode Decoding 357

7.1 Motivational example: different fruits forming distinct clusters in a feature space. 362
7.2 Overview of the classic image classification *pipeline*: descriptor extraction (LBP, HOG), formation of the feature space, and classification via k-NN. Not applicable to Deep Learning models (CNNs, YOLO), which learn *features* end-to-end directly from pixels. **Source:** prepared with the aid of Gemini Notebook ({GOOGLE}, 2025). 363
7.3 Visual comparison of different descriptors applied to the same image. Each descriptor reveals distinct aspects of the scene. 365
7.4 Histogram of LBP codes from the cameramen image. Each bar represents the relative frequency of an LBP code, forming the feature vector used to describe the texture. 366
7.5 First 100 components of the HOG feature vector. 368
7.6 Importance of feature normalization for the k-NN classifier. 370
7.7 Concept map of the image classification process using classical descriptors (LBP, HOG, raw pixels) and a traditional classifier (k-NN). It does not apply to Deep Learning models (CNNs, YOLO), which learn *features end-to-end* directly from pixels. 372
7.8 Confusion matrices obtained by the k-NN classifier using three different descriptors. The rows represent the actual class (Apple, Banana, and Orange) and the columns the predicted class. The greater the concentration of values on the main diagonal, the better the descriptor's performance. 374
7.9 Detailed comparison of global accuracy in a realistic scenario. Note how structurally extracted descriptors (LBP and HOG) outperform the use of pure raw pixel intensities. 375
7.10 Illustration of the synthetic data generation process with *make_classification*. On the left, visualization of the three classes in a two-dimensional space formed by two informative features. On the right, relative importance of each feature for classification, showing that only 7 of the 10 features are effectively discriminative, while the others are redundant (2) or duplicated (1). 378
7.11 Classification of a new sample by the k-NN algorithm. The test point (star) is classified using the three nearest neighbors, highlighted by circles. 382
7.12 Interactive simulator of the k-NN decision boundary: add training points and adjust the value of k to observe the effect on the decision region. 383
7.13 Sample of handwritten digits from the *load_digits* dataset, used as a classification case study. 384
7.14 Confusion matrix of the k-NN classifier trained with raw intensity vectors (pixels). 386
7.15 Comparison of accuracy between raw pixel descriptors and HOG for the k-NN classifier (k=3) on the digits dataset. 387
7.16 Mean accuracy by cross-validation (5 folds) as a function of parameter k, for the digit base with raw intensity vectors. 389

7.17 Accuracy on the training and test sets for different values of k. The colored regions represent, conceptually, behavior trends of the classifier: higher risk of overfitting (red), bias-variance trade-off (green) and higher risk of underfitting (blue). . . . 390
7.18 Synthetic samples of the three texture classes used in the classification experiment, generated with Gaussian noise and intra-class variability. . . . 392
7.19 Average accuracy obtained via cross-validation (5-fold) for the descriptors of Raw Pixels, LBP, and HOG applied to the synthetic textures dataset. . . . 394
7.20 Detailed evaluation metrics for the k-NN classifier with LBP descriptors . . . 397
7.21 EP07_01 Simulator: Step-by-Step k-NN Classifier . . . 403
7.22 EP07_02 Simulator: Effect of *Z-score* Normalization on k-NN Distance . . . 405
7.23 Simulator EP07_03: Precision x Recall . . . 407
7.24 EP07_04 Simulator: LBP Code of a 3x3 Neighborhood . . . 409
7.25 EP07_05 Simulator: HOG Histogram of a Cell (mapping angles to bins) . . . 412
7.26 Simulator EP07_06: k-NN Multi-Class *Pipeline* (voting, tie-breaking and confusion matrix) . 415
7.27 Reference mosaics (ASCII PGM format) used in EP07_07. Case 1: 2x2 grid with one sample of each class. Case 2: 3x3 grid with repeated classes and greater variability. . . . 420
7.28 Simulator EP07_07: Classification of a Texture Mosaic via LBP + k-NN . . . 420

8.1 Overview of the main concepts covered in this chapter, including feature matching, object detection, image segmentation, and their relationships with modern approaches based on *Deep Learning*. **Source:** prepared with the assistance of Gemini Notebook ({GOOGLE}, 2025). . . 424
8.2 Comparison of performance of different implementations of the Hamming distance. . . . 427
8.3 Interactive simulator of the ORB descriptor: explore the rotation logic and binary descriptor construction. Change the rotation to observe how the BRIEF binary test sampling pattern (green and red lines) dynamically orients itself to ensure angular invariance. . . . 428
8.4 Original image generated by Gemini and rotated version (20°), simulating a change of viewpoint. 429
8.5 The 5 best ORB feature matches between the original image and the synthetic scene, before RANSAC filtering. . . . 431
8.6 Interactive simulator of the RANSAC algorithm: adjust the distance threshold and run the algorithm to observe the separation between *inliers* and *outliers*. . . . 432
8.7 Interactive simulator of RANSAC applied to image registration: key points from two images are matched by a descriptor, some correspondences are spurious (*outliers*), and RANSAC estimates the similarity transformation that aligns most of them. . . . 433
8.8 Correspondences classified as *inliers* (green) by RANSAC when estimating the homography between the two images. . . . 434
8.9 Recording the scene image using the inverse homography estimated by RANSAC. . . . 435
8.10 Face and eye detection with *Haar Cascade*. . . . 438
8.11 Effect of Non-Maximum Suppression (NMS): multiple redundant detections (left) are reduced to one detection per object (right). . . . 440
8.12 Classic segmentation (not deep learning-based): semantic mask (coin vs background) via Otsu thresholding, and instance segmentation via connected-component labeling. . . . 442
8.13 Separation of touching coins via distance transform + *watershed*, without erosion of the coin contours. . . . 444
8.14 *Bounding boxes* reconstructed from the YOLO annotations file exported by *mm.saveMeasures*, overlaid on the original image of the coins. . . . 448
8.15 EP08_01 Simulator: Hamming Distance Between Two Binary Descriptors . . . 456
8.16 EP08_02 Simulator: RANSAC — Voting by Inliers among Candidate Models . . . 458
8.17 EP08_03 Simulator: Rectangular Sum in O(1) — Multiple Edge Cases . . . 460
8.18 EP08_04 Simulator: IoU and Non-Maximum Suppression . . . 462
8.19 Simulator EP08_05: Connected Components Labeling — 4 vs. 8 Connectivity . . . 464
8.20 Simulador EP08_06: Extraction of *Bounding Boxes*, Centroids and Properties with *mm.measure* 466
8.21 EP08_07 Simulator: Morphology with Configurable Connectivity and Measurement . . . 468
8.22 EP08_08 Simulator: Median Filtering in Grayscale and Object Measurement with OpenCV . 470
8.23 Simulator EP08_09: Illumination Gradient, Adaptive Threshold and OpenCV Measurement 471

8.24 EP08_10 Simulator: Separation of Tangent Disks via L2 Distance Transform and *Watershed* 473
8.25 EP08_11 Simulator: Geometric Classification with Independent Controls for *BBox Overlays* (GT and DET) . . . 475
8.26 Simulator EP08_12: Image used in EP08_12. On the left, the original image. On the right, the image with the *bounding boxes* from the answer key file. . . . 478
8.27 EP08_12 Simulator: Segmentation Accuracy in Multiple Objects (IoU) . . . 479

9.1 Overview of the main concepts addressed in this chapter. **Source:** prepared with the aid of *Gemini Notebook* ({GOOGLE}, 2025). . . . 482
9.2 Interactive simulator of classic 2D convolution: choose among the three synthetic 12×12 input images (house, happy or sad face) and a 3×3 filter (Sobel V, Sobel H, Sharpen, or Identity) and step forward to observe how the local inner products of the receptive field build the feature map cell by cell. . . . 485
9.3 Interactive simulator of the Artificial Neuron in a CNN context: toggle between a convolutional layer neuron (where x_i are pixel intensities in a receptive field and w_i are kernel weights) and a fully connected layer neuron, adjusting inputs, weights, bias, and activation function to visualize the computation of z and output y in real time. . . . 487
9.4 Interactive Convolutional Layer Simulator: choose among three 12×12 input images (house, happy face, or sad face) to observe how the same fixed *kernels* react to different edges and shapes. Navigate the receptive field with the buttons or automatic playback, adjust the activation function and toggle zero-padding, while the feature map is revealed cell by cell, with detailed term-by-term calculation and the output dimension formula in real time. . . . 489
9.5 Interactive simulator of the ReLU activation function: drag the control to see how negative values are zeroed out and positive values are preserved, both in the curve and in a real feature map. . . . 490
9.6 Interactive *pooling* simulator: choose between *max-pooling* and *average-pooling* and step forward to observe the reduction of the spatial resolution of the feature map. . . . 490
9.7 Interactive simulator of the Loss Function (*Cross-Entropy*): select the real class of the image (House, Happy or Sad) and adjust the probabilities estimated by Softmax to visualize the calculation of the scalar penalty and the graph of the negative logarithm in real time. . . . 491
9.8 Interactive *Backpropagation* simulator: advance through the Chain Rule steps to follow the error signal flow in the reverse direction of the network, observing the computation of partial derivatives of the gradient at each layer. . . . 492
9.9 Interactive simulator of Optimization Algorithms: compare the trajectory of SGD and Adam over a non-convex loss surface with a heatmap and contour lines. The solid line shows the selected optimizer advancing step by step; the dashed line shows, for instant comparison, the full path that the other optimizer would take from the same starting point. Notice how SGD gets stuck in the Local Minimum on the right, while Adam may or may not cross the central barrier depending on accumulated momentum and learning rate. Click anywhere on the plot to reset the starting point of the weights. . . . 493
9.10 Simplified architecture of a CNN for image classification, highlighting the *forward pass* and the training steps through the loss function, backpropagation, and the optimization algorithm. 494
9.11 Interactive simulator of a CNN architecture: choose one of the 12×12 input images (house, happy face, or sad face), click on each block of the *pipeline* — Input, *Conv+ReLU*, *Pooling*, *Flatten*, *FC*, and *Softmax* — and run real training steps (*forward pass* + backpropagation) to observe the loss and accuracy evolving, the *kernels* being adjusted, and the *Softmax* starting to point to the correct class. . . . 495
9.12 Interactive simulator for computing the gradient of a weight from the convolutional *kernel*. . 498
9.13 Digit samples from the training set after conversion to PyTorch *tensors* and normalization. . 501
9.14 Representation of the flow of dimensional transformations of the tensors throughout the *CNNDigitos* architecture. . . . 501
9.15 Training and evaluation curves of the CNN on the digits dataset: evolution of cross-entropy loss on the training set and accuracy on the test set over 50 epochs. . . . 505

9.16 Flow of activations of the trained CNN when processing a real example of the digit 3, from the *load_digits* dataset: tensor dimensions layer by layer, from input to output *logit*. 506
9.17 Computational graph of *CNNDigitos* generated via *torchviz*, showing the *forward* operations and the gradient nodes (*backward*) associated with each trainable parameter. 507
9.18 Detailed graph of the CNNDigitos class generated via *torchviz*. 509
9.19 Comparison of accuracy between the classic classifiers from Chapter 7 (raw pixels and HOG with k-NN) and the CNN trained in this chapter, on the same digit dataset. 510
9.20 Samples from the training sets after preprocessing and dimensionality adjustment to PyTorch tensors: Domain A (digits 0 to 4, source task) and Domain B (digits 5 to 9 reindexed to 0 to 4, target task). 513
9.21 Comparison of accuracy on the Domínio B test set (digits 5 to 9) under data restriction (20 training examples): demonstration of the impact of rigid freezing and negative transfer in low-capacity networks. 517
9.22 Flow of activations and dimensional transformations of the tensors in the transfer learning model when processing a test sample from Domain B (digit 7, reindexed to class 2). 518
9.23 519
9.24 521
9.25 521
9.26 Synthetic samples from the leaf diagnosis *dataset*: healthy leaf, diseased leaf (dark spots) and unknown symptom (yellowish chlorosis) with salt-and-pepper noise. 525
9.27 527
9.28 528
9.29 530
9.30 Initial samples of the noisy synthetic geometric objects *dataset* with YOLO-format bounding boxes overlaid. 533
9.31 Comparison between the original image with salt-and-pepper noise and the image after applying the Median Filter (3x3). The *bounding boxes* in YOLO format remain unchanged. 535
9.32 538
9.33 540
9.34 Samples from the synthetic nodule dataset: grayscale image under noise and its respective reference binary mask shown. 542
9.35 Classical segmentation baseline: smoothing, Otsu thresholding, and morphological opening displayed. 543
9.36 Evolution of compact U-Net training: loss reduction and increase in the average IoU index on the validation set over 35 epochs. 547
9.37 Quantitative comparison (mean IoU) and qualitative comparison between the classical approach and the U-Net trained on three validation samples. 548
9.38 551
9.39 552
9.40 553
9.41 Augmented reality based on ArUco marker: marker inserted in real scene and rotated, detected and replaced by virtual image via homography. 555
9.42 Measurement of real physical dimensions using a reference card with a known scale (8.56 cm) rendered. The gray rectangle represents the reference card and the blue rectangle represents the target object. The estimated dimensions of the object are displayed in centimeters. 558
9.43 EP09_01 Simulator: Manual 2D Convolution (cross-correlation + bias + ReLU, with adjustable *padding* and *stride*) 566
9.44 Simulator EP09_02: Manual Pooling (max vs. mean, with adjustable window k and stride s) 569
9.45 Simulator EP09_03: Parameter Counting — Convolution vs. Fully Connected Layer 574
9.46 EP09_04 Simulator: IoU and Non-Maximum Suppression (NMS) 576
9.47 Simulator EP09_05: Segmentation Evaluation — IoU and Dice Pixel by Pixel 579
9.48 EP09_06 Simulator: Integrated Pipeline — From Real-World Detection to Measurement . . 582

A.1 Chessboard question generated by MCTest, illustrating the parameterized statement with the `height` value drawn for the variation and the input/output example of the first test case. . . 588
A.2 Real question generated by MCTest for Mock Exam 4 — "Morphological Operators" topic. . 590

List of Tables

1 Different publication formats automatically generated from the same source. 4
2 PDF page count and rendering time per combination (real average parallelism of ~5 processes, peak of 8). The base Python/Portuguese combination only renders the outputs already stored in the source *notebooks*; the others re-execute all the code. 4

1.1 Connection among IP, CV, and other scientific areas. 5
1.2 Main types of digital images and their respective sets of possible values for each pixel. 8
1.3 Main libraries and modules used throughout this book for matrix computation, DIP, CV, visualization, and validation of exercises. 9

2.1 Scale and channel order conventions in major DIP libraries. 50
2.2 Comparison between packing strategies and processing efficiency. 56
2.3 Comparison of distance metrics applied to the pixel grid. 58
2.4 Main digital image storage formats and their applications in DIP. 58
2.5 Interpolation methods available in mm.resize and their respective numbers of neighbors used in the calculation. 63

3.1 Histogram equalization algorithm. 99
3.2 Typical interpretation of *kernel* coefficients. 108
3.3 Steps of *Unsharp Masking*. 121
3.4 Mean vs. median with one corrupted pixel. The median ignores the outlier; the mean is shifted by ~40 levels. 124

4.1 Preprocessing techniques evaluated to improve the separation between coins and background before applying the Otsu method. 156
4.2 Properties of opening and closing. 167
4.3 Sequences of the alternating sequential filter mm.asf. 168
4.4 Comparison between the *clohole* and *edgeoff* operators. 175
4.5 Simplified taxonomy of the main segmentation and refinement approaches studied in this chapter. 183
4.6 Complete marker-based *watershed* pipeline for real images. 195
4.7 Simplified pipeline used in the didactic example of Figure 4.25. 195
4.8 Comparison between approaches based on connected components and contours. 198

5.1 Correspondence between the regions of the magnitude spectrum after applying the FFT Shift. 232
5.2 Comparison of the complexity of direct convolution, separable convolution, and filtering via the Fast Fourier Transform (FFT). 238
5.3 Subbands produced by the 2D Discrete Wavelet Transform (DWT), indicating the filters applied in each direction and the predominant content of each component. 250
5.4 Comparison among wavelet families, highlighting support length, number of vanishing moments, symmetry, and typical applications. 251
5.5 Comparison between the Discrete Fourier Transform (DFT) and the Discrete Wavelet Transform (DWT), highlighting their main characteristics and applications. 259
5.6 Categories of redundancy in digital images and their respective exploration mechanisms. . . . 260
5.7 Stages of the JPEG compression pipeline and their respective design rationales. 264
5.8 Structural comparison between the main rasterized image formats. 268
5.9 Synthesis of the stages of the JPEG compression *pipeline* and their respective impacts. 275

6.1 skimage.data images used in the experiments of this chapter. 300
6.2 Selection of representative public images available in the *skimage.data* module, organized by application area. 300

7.1 Set of chromatic, textural, and geometric descriptors used to represent fruit images. 376
7.2 Simplified example of fruit classification using two normalized features. 379

8.1 Comparison between the main visual segmentation paradigms. 441
8.2 Representative Deep Learning architectures for image detection and segmentation. 449

9.1 Comparison among the main computer vision tasks according to the granularity of the information produced. 482
9.2 Comparison among the three transfer learning scenarios presented in this section. 528

C.1 URL filter expressions used in the SEB configuration. 593

DIP and CV — Python

Welcome to the PDI and Computer Vision textbook — Python / English version.

Organization

- **Part I — PDI Fundamentals** — Representation, histograms, filtering, morphology
- **Part II — Computer Vision** — Segmentation, descriptors, detection, deep learning

Preface

This book is a **work in permanent progress** in the fields of **Digital Image Processing (DIP)** and **Computer Vision (CV)**, conceived as interactive instructional material for undergraduate and graduate courses in Computer Science, Engineering, and related fields.

> **!** Highlight
>
> The work is grounded in the methodology described in Zampirolli et al. (2025) — an extension of Zampirolli et al. (2024), an award-winning paper in the **Educational Resources and Environments** track of **EduComp 2024**. The content integrates the `morph.py` library, developed by the author.

This is not a static book. Its content evolves continuously as examples are refined, new sections are added, and pedagogical approaches are improved based on usage experience and feedback from students and instructors. The work is thus treated as a *project in constant evolution*, aiming to keep pace with both technological advances and best teaching practices in DIP and CV.

Context of the first edition

The content of this edition was developed between May and August 2026, during the first offering of the Digital Image Processing course based on this instructional material. The course was taught to two undergraduate sections — composed mostly of Computer Science students, in the morning period — and one graduate section in Computer Science, all at UFABC. This first edition represents the outcome of that initial offering, consolidating the content developed, tested, and continuously refined throughout the term.

The teaching methodology followed a structured sequence in each class. First, a short video, averaging 7 minutes, generated with **Gemini Notebook**, was shown, alternating between presenting the conceptual part of the chapter and solving the Programming Exercises (EPs). In the latter case, nearly all EPs were solved during the class itself. Next, a set of about 12 *slides*, also generated with **Gemini Notebook**, was presented, using the book's complete PDF and the `morph.py` file as sources. These *slides* were produced from a specific *prompt* for generating the theoretical and practical content of each chapter.

After this initial stage, about 100 minutes of class time remained for exploring the chapter's two Colab *notebooks* — one theoretical and one practical — with an emphasis on the interactive simulators and running the code blocks, allowing students to modify parameters and observe their effects. In classes held in the lab, students transferred the answers to the EPs developed in Colab directly to the Moodle VPL activities, where they were automatically graded. This publication and use process for the EPs is presented at the end of this preface.

Ongoing assessment included biweekly practice tests, administered with the **SEB (*Safe Exam Browser*)** in Moodle, containing EPs similar to those in the exercise sets. Each practice test granted a 5% bonus on the final grade. The course's two exams also used SEB, but were composed of parameterized questions generated in **MCTest**, whose description combines LaTeX and parameters in the `[[code:variable]]` format, defined in Python snippets delimited by `[[def: ... ]]` within the question itself. This process made it possible to generate 110 exam variations, individually assigned to students at random.

Appendix A details the creation of these questions in MCTest and their export to Moodle; Appendix B describes the configuration of the VPL activities, including the process of publishing the EPs; Appendix C presents the SEB configuration for the practice tests and exams; Appendix D describes the generation of the videos and *slides* used in class with **Gemini Notebook**; and Appendix E details the use of **Artificial**

Intelligence (AI) in generating Socratic *feedback* for formative and summative activities, complementing the automatic grading performed by VPL.

How this book is produced

The content of this book is developed in **Quarto** and stored in the `all` folder of the **github.com/fzampirolli/pdi-vc** repository. From a single source, the book is automatically generated and published in different formats, shown in Table 1.

While the PDF edition records the state of the work at the end of the course's first offering in 2026, development of the book continues on an ongoing basis. Every update to the GitHub repository automatically generates new versions of the HTML pages, the PDF, and the notebooks, immediately making improvements available to readers.

Table 1: Different publication formats automatically generated from the same source.

Format	Description
HTML	*Web* version, with interactive simulators and navigation between chapters: fzampirolli.github.io/pdi-vc/
PDF	Version suited for printing or *offline* reading: livro.pt.py.pdf
Notebooks (`.ipynb`)	Compatible with **Jupyter** and **Google Colab**, allowing readers to run, modify, and experiment with code examples and Programming Exercises (EPs). A custom filter preserves cross-references, figure and table numbering, and automatically formats citations according to the ABNT standard.

The work is published in **ten combinations** — each code track (**Python** and **C++**) in five languages (Portuguese, English, French, Spanish, and Italian). With the translation cache already populated, a full regeneration (translation, re-execution of the *notebooks*, HTML and PDF rendering, and publication) takes about **72 minutes**, with a real average parallelism of ~5 processes (peak of 8); the **first** generation of a new language, with an empty cache, is much slower — about **80 minutes** per combination, as observed for the first generations of Spanish and Italian. Table 2 summarizes each combination (with the cache already populated): PDF page count and rendering time. This wall-clock total is much lower than the sum of each individual combination's time (over **5 hours** if run one at a time): on a multi-core CPU, several combinations run at the same time, so the total time is not simply the sum of the individual ones.

Table 2: PDF page count and rendering time per combination (real average parallelism of ~5 processes, peak of 8). The base Python/Portuguese combination only renders the outputs already stored in the source *notebooks*; the others re-execute all the code.

Combination	Track	Language	Pages	Rendering time
`py.pt`	Python	Portuguese (base)	654	~7 min
`py.en`	Python	English	644	~31 min
`py.fr`	Python	French	666	~31 min
`py.es`	Python	Spanish	666	~31 min
`py.it`	Python	Italian	658	~31 min
`cpp.pt`	C++	Portuguese	434	~26 min
`cpp.en`	C++	English	426	~34 min
`cpp.fr`	C++	French	434	~38 min
`cpp.es`	C++	Spanish	430	~37 min
`cpp.it`	C++	Italian	428	~35 min

Validation status. Only the `py.pt` combination (Python, Portuguese) is the curated source: it is where the author writes, reviews, and validates all the content. The other nine combinations — the **C++** tracks and the

translations into English, French, Spanish, and Italian — are **generated automatically**, by language-model translation and code transpilation, and still **require detailed review**. The main concern is the **text embedded in some figures**: where the translated version has not yet been produced, the image appears with Portuguese text (each translated figure follows the same language suffix as the other generated files, e.g. `image_en.png`).

The HTML pages, the PDF, and the *notebooks* available in the project always reflect the most recent state of development. When an **official edition** is published, it is identified by a *tag* in the GitHub repository, allowing the content corresponding to that edition to be reproduced exactly. This way, readers can follow both the continuous evolution of the project and retrieve any previously published official edition.

```
git clone https://github.com/fzampirolli/pdi-vc.git
cd pdi-vc
git checkout <version-tag>
```

Each chapter provides two **Run on Colab** buttons (Figure 1), which link to complementary environments:

1. **Theoretical Part**, located at the beginning of each chapter, containing the concepts presented and executable code examples;
2. **Practical Part**, in the **Practical Part with Programming Exercises** (EPs) section, dedicated to the exercises and their interactive simulators.

Figure 1: Clickable **Run on Colab** button, available at the beginning of the **Theoretical** and **Practical** parts of each chapter, allowing interactive execution of the examples and exercises. In the PDF version, there is a direct HTML *link* between the simulator image and its caption.

Notebook generation is fully automated by the `gerar_notebooks_alunos.py` *script*, which adapts the content for interactive environments, allowing it to run without installing Quarto.

Use of Artificial Intelligence tools

The conception, pedagogical design, conceptual structure, and core content of this book are of the **author's exclusive authorship**.

In the editing and development-support process, free-tier versions of **Artificial Intelligence (AI)** tools such as **ChatGPT**, **Claude**, **DeepSeek**, and **Gemini** were used, strictly as auxiliary resources. Their use was limited to supporting the review and improvement of textual style, optimizing code syntax, and assisting in generating conceptual illustrations and examples.

All responses and content suggested by these tools were subject to **critical analysis, verification, technical validation, adaptation, and integration by the author**, who takes responsibility for the text, the code, and the instructional resources presented. AI tools are **not considered authors or co-authors of the work**, nor responsible for the intellectual, technical, or pedagogical decisions underlying its content.

It should also be noted that the **simulators** and interactive resources in the book — available in the HTML and Jupyter Notebook (IPYNB) versions — were designed and implemented as part of the work's pedagogical approach, aiming to provide direct experimentation with the concepts presented and to foster readers' learning autonomy.

Distinct from the editorial use described above, the multi-language generation *pipeline* (see "How this book is produced") employs a language model as a **system engineering component**: automatic translation of content between Portuguese and other languages, with an incremental cache that avoids retranslating unchanged passages. This step uses the paid **DeepSeek** API (`deepseek-v4-flash` model); the complete book has already been translated from Portuguese into **English**, **French**, **Spanish**, and **Italian**, and

Chapters 1 to 5 additionally have a **C++** track that truly compiles and runs — at a cumulative cost on the order of a few dollars, thanks to the incremental cache.

The `morph.py` library

The `morph.py` library (Zampirolli et al., 2025) accompanies the whole work as a teaching-support tool. Its goal is not to replace established libraries such as **OpenCV**, **scikit-image**, **scikit-learn**, **NumPy**, or **Matplotlib**, nor to compete with them in performance or scope. Instead, it seeks to **make the fundamental algorithms of Digital Image Processing and Computer Vision (DIP-CV) transparent**, allowing students to understand their implementation, modify the code, and develop new functionality.

Whenever possible, the library offers two versions of the same algorithm: a **didactic implementation**, written step by step to directly reflect the mathematical definitions presented in the book, and an **optimized implementation**, based on specialized libraries for practical applications.

> **ℹ Technical Heritage**
>
> The structure of `morph.py` is based on Mathematical Morphology tools developed in Brazil, such as **MMachLib** (Lotufo et al., 1997) and **MMach** (Barrera et al., 1998), as well as **mmorph**, used in the work of Dougherty; Lotufo (2003). These libraries served as references for teaching in the field for decades.

This philosophy can be observed, for example, in the morphological dilation operators:

- `mm.dil`: optimized implementation, suited for practical applications;
- `mm.dil0`: didactic implementation of dilation for **planar structuring elements**, following directly the classic definition of Mathematical Morphology;
- `mm.dil1`: didactic implementation of dilation for **structuring functions (non-planar *kernels*)**, rigorously following its mathematical formulation.

The library also provides functions for reading, displaying, color conversion, filtering, and mathematical morphology through a simple interface:

```
from morph import mm

img = mm.gray(mm.read("lena.jpg"))  # read and convert to grayscale
grad = mm.gradm(img)                # morphological gradient
mm.show(grad)                       # display
```

Throughout the book, new methods are introduced into the library to simplify solving the Programming Exercises (EPs). Among them are `mm.readTrain()` and `mm.readTest()`, which automate reading the training and test datasets used in the pattern recognition chapters.

In addition, all **Gemini Notebook** projects include the `morph.py` file, allowing direct consultation and discussion of the implementation of the algorithms presented in the book. This way, students can use Gemini Notebook itself as a study assistant, asking questions such as:

> Explain how the `mm.knn0` algorithm from `morph.py` was implemented, showing the code and commenting on each step in a didactic way. Also explain the differences between `mm.knn0()` and `mm.knn()`.

> **❗ Didactic implementations and optimized implementations**
>
> In several chapters, the same algorithm is presented in two versions. Functions with a suffix `0`, `1`, etc. (for example, `mm.knn0()`, `mm.dil0()`, and `mm.dil1()`) are didactic implementations, developed to facilitate understanding of the algorithms. Functions without a suffix (such as `mm.knn()` and `mm.dil()`)

use optimized implementations, often based on specialized libraries such as OpenCV, scikit-image, and scikit-learn, and are better suited for practical applications.
In activities graded by **Moodle's VPL**, libraries such as **scikit-learn** and, in some cases, **scikit-image**, cannot be used due to memory limitations in the execution environment. For this reason, the Programming Exercises (EPs) preferably use the didactic implementations from `morph.py`, which produce the same results, albeit with lower computational performance. This restriction does not apply when programs are run locally, for example in **VS Code**, **Jupyter Notebook**, or **Google Colab**.

The library's source code is available at:

https://github.com/fzampirolli/pdi-vc/tree/master/morph

Programming Exercises (EPs) and Automatic Grading

Each unit of the book includes practical **Programming Exercises (EPs)** of increasing complexity, designed to consolidate the concepts presented throughout the chapter. Each EP is accompanied by an **interactive simulator**, available in both HTML and IPYNB versions, which lets students manipulate parameters, visualize the algorithms' behavior, and develop an intuitive understanding of the problem before starting implementation. This way, the learning process combines experimentation, programming, and automatic grading.

Solutions are validated locally by the `TestSuite` class (`testsuite.py`), which compares the program's output against the test case files (`.cases`):

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

The system supports multiple languages (Python, Java, C++, C, JavaScript, and R) and can be integrated directly with Moodle. Instructors interested in the complete step-by-step process for publishing EPs as VPL activities should consult the **Instructor's Guide** at the end of this preface, and Appendix B.

Open source

The project is governed by open-source principles. The public repository gathers the text, code, images, and *scripts*: **github.com/fzampirolli/pdi-vc**

Each version of the book is permanently archived on **Zenodo** with a citable *DOI*. To cite this material in academic work:

> ZAMPIROLLI, Francisco de Assis. **PDI+VC — Digital Image Processing and Computer Vision**. UFABC, 2026. DOI: 10.5281/zenodo.20784605

It should be noted, for the record, that the content presented in this work reflects the critical outlook, pedagogical approach, and teaching experience of its author. Accordingly, the analyses, approaches, and opinions expressed throughout this book represent solely the understanding of its creator, and do not constitute or reflect an official or institutional position of the Federal University of ABC (UFABC).

Before you begin: Python *Notebooks*

The concept of ***Literate Programming***, proposed by Donald Knuth (Knuth, 1984), underlies the structure of this material. The logic reverses the traditional paradigm: the program is written for human reading, resembling an essay, while the code is extracted separately for computational execution.

The content is structured into *notebooks* — documents that interleave **text cells** (in Markdown) and **code cells** (in Python).

- **Execution:** code cells are identified by `[ ]`. They can be run with `Shift + Enter` or the ▷ button in the interface.
- **Environments:** the *notebooks* can be run locally, through Jupyter or Visual Studio Code (VS Code), as well as in cloud environments such as Google Colab.

> Note on the format
>
> In interactive environments, the code can be modified and executed. In the static versions (HTML or PDF), the code blocks are for reading and reference purposes, without any loss to the completeness of the explanations.

Instructor's Guide: Publishing EPs in Moodle

This section is intended for instructors who wish to integrate the Programming Exercises (EPs) with Moodle's ***VPL (Virtual Programming Lab)*** *activities, complementing Appendix B.*

Integration with Moodle (VPL)

The EPs from the *notebooks* can be converted into **VPL** activities in Moodle. The `ep_tools.py` *script* (run via `make build`) automates exporting the EPs from the end of each chapter in two steps:

1. **Extraction** (`make eps`): generates a standalone interactive HTML per EP, with the statement, examples, and simulator, in `gen/book/eps/<version>/`. See the full list at: https://fzampirolli.github.io/pdi-vc/eps/py.pt/index.html
2. **Conversion** (`make moodle`): transforms the HTML into a self-contained fragment, with no external CSS dependency, ready to be pasted into the Moodle editor, in `gen/book/eps/<version>_moodle/`. See an example at: https://fzampirolli.github.io/pdi-vc/eps/py.pt_moodle/EP01_01.html

Additionally, all the interactive simulators developed throughout the book can be accessed directly at https://fzampirolli.github.io/pdi-vc/simuladores/py.pt/.

> Other tracks and languages
>
> The links above use `py.pt` as an example. To access EPs or simulators for another language/locale combination, just swap that segment in the URL — for example, `cpp.it` for the C++ track in Italian: `https://fzampirolli.github.io/pdi-vc/eps/cpp.it/index.html`. The structure is the same for every available combination, under `/eps/<version>/`, `/eps/<version>_moodle/`, and `/simuladores/<version>/`.

When publishing with `make publish`, both **versions** of each EP become available at the links indicated. Instructors can combine both strategies: paste the Moodle version into the VPL editor (with a working simulator) and include a *link* in the statement to the full version, where formulas are rendered correctly by MathJax.

> Mandatory review before publishing to Moodle
>
> Although automated, the conversion requires instructor review due to limitations of Moodle's TinyMCE/HTML Purifier editor:
>
> - **Mathematical formulas** — TinyMCE discards backslashes (\), corrupting LaTeX equations. For this reason, the Moodle version converts simple formulas to plain HTML (entities such as `≥`, `×`). Complex formulas (`\begin{cases}`, matrices, integrals) may come out incomplete — review them and, if necessary, rewrite them as text or point to the full version via *link*.
> - **External figures** — Supplementary images must be inserted manually into the VPL activity.

- **Simulators** — Work perfectly as long as they use standard JavaScript with *inline* styles and direct DOM manipulation (`getElementById`). **Warning:** if you include LaTeX formulas containing the \ character in Moodle, the simulators may stop working.

How to Publish to Moodle

1. Access the Moodle version of the desired EP:

   ```
   https://fzampirolli.github.io/pdi-vc/eps/py.pt_moodle/EPXX_YY.html
   ```

2. Copy the full source code of the page (`Ctrl+U` → `Ctrl+A` → `Ctrl+C`).
3. In Moodle, create the VPL activity, open the HTML editor, paste the content (`Ctrl+V`), and save.
4. Import the `.cases` files from the `all/capXX/casos/` folder into the VPL test settings.

Reusing Test Cases

The `.cases` files are compatible with VPL, allowing the same test set to be used both in local development and in automated grading in Moodle.

Part I

Part I — PDI Fundamentals

Chapter 1

Fundamentals and First Steps

Run Colab Open GitHub

This chapter inaugurates **Part 1** of the book, dedicated to the fundamentals of Digital Image Processing (DIP). The mathematical representation of digital images and the main methods for their manipulation are presented.

Python language and the `morph.py` library (ZAMPIROLLI, 2025) are used.

1.1 Objectives

At the end of this chapter, you will be able to:

- Understand the physical and mathematical nature of the digital image $f(x, y)$.
- Identify the bands of the electromagnetic spectrum relevant to DIP.
- Perform basic operations: reading, displaying, and saving images.
- Access and modify pixel intensities individually.
- Apply manual thresholding.
- Set up the Python development environment.
- Handle array structures (NumPy) without falling into memory traps.

1.2 Before you begin: Interactive *Notebooks*

This material was built under the concept of ***Literate Programming***, conceived by Donald Knuth in the 1980s (KNUTH, 1984). Knuth — also the creator of the **TeX** system for digital typesetting — proposed that programs be written as a logical narrative for human beings, interleaving code and documentation.

To run a cell, press Shift + Enter or click the ▷ button.

> **Note on the format**
>
> In rendered versions (PDF or HTML), the code is presented in static blocks for reading and reference purposes. Interactive execution requires access via Google Colab (available at the top of the page) or in a local environment via VSCode or Jupyter Notebook.

1.3 Fundamentals

The study of image-based systems encompasses an ecosystem of integrated disciplines that transform raw visual data into structured knowledge. While some areas focus on generating representations, others are dedicated to processing and analyzing this data to support complex technological applications.

The diagram presented in Figure 1.1 establishes the distinction and complementarity between Digital Image Processing (DIP) and Computer Vision (CV). DIP, highlighted in **green**, focuses on image-to-image transformation, aiming at quality improvement or preprocessing, such as noise removal and contrast enhancement.

In contrast, CV, marked in **blue**, focuses on interpreting visual content to extract models or information, such as object and gesture recognition. The intersection region illustrates the synergy between the areas, where DIP prepares the visual data for interpretation by CV. The map also demonstrates the interconnections of both disciplines with areas such as Robotics, Computer Graphics, Artificial Intelligence (AI), and Neuroscience.

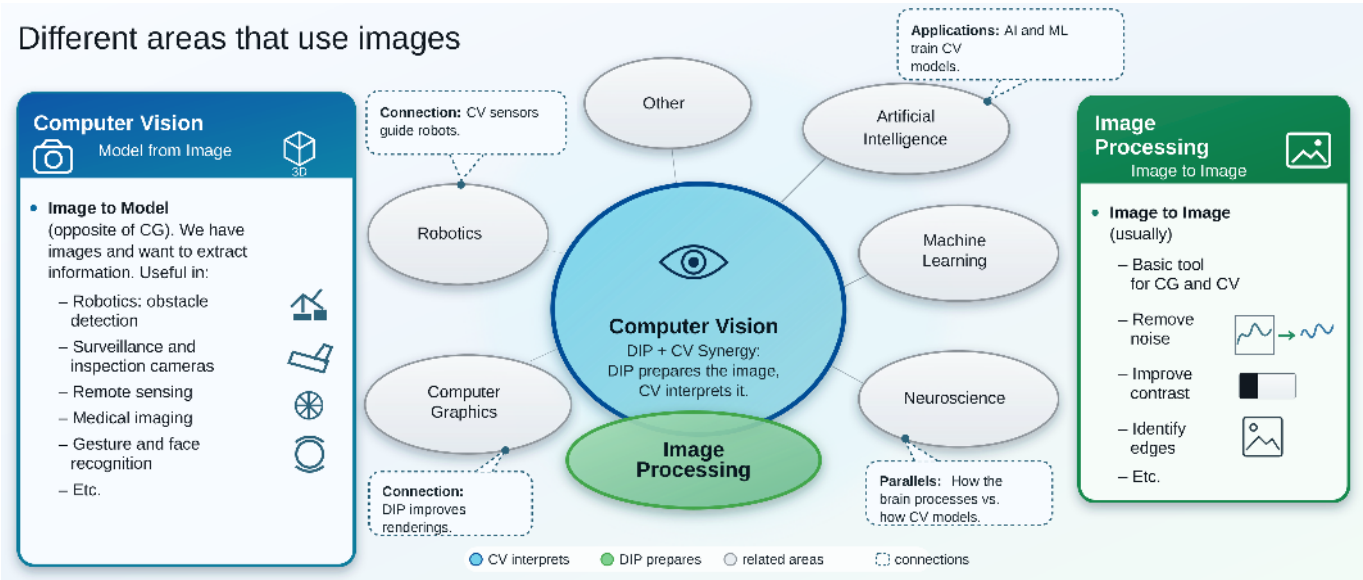


Figure 1.1: Relational diagram detailing the fundamental distinctions, synergies, and interconnections between DIP and CV in the context of image-based systems.

1.3.1 Computer Vision

- **Focus:** *Image* → *Model* (inverse path of Computer Graphics).
- **Goal:** Extract high-level information from images or videos.
- **Typical applications:**
  - Robotics – obstacle detection, localization, and autonomous navigation.
  - Surveillance and inspection – event recognition, license plate reading, quality control.
  - Remote sensing – satellite image analysis, environmental mapping.
  - Medical imaging – tumor detection, organ segmentation, diagnostic support.
  - Human–computer interaction – gesture recognition, facial expression analysis, eye tracking.
- **Relationship with other fields:** leverages Machine Learning and AI techniques to classify and interpret scenes; serves as the "eyes" of Robotics.

1.3.2 Digital Image Processing (DIP)

- **Focus:** *Image* → *Image* (typically—transformation of one image into another).
- **Objective:** Enhance visual quality or extract low-level features.
- **Common applications:**
  - Noise removal (mean, median, and Gaussian filters).
  - Contrast enhancement (histogram equalization, gamma adjustment).
  - Edge detection (Sobel, Canny, Laplacian).

  - Segmentation (thresholding, region growing, *watershed*).
  - Geometric transformations (resizing, rotation, perspective correction).
- **Relationship with other areas (see Table 1.1):**
  - It is the **foundation** for most CV systems (preprocessing).
  - Computer Graphics often applies DIP for postprocessing (e.g., smoothing, enhancement).
  - AI techniques can optimize processing parameters (e.g., filter learning).

Table 1.1: Connection among IP, CV, and other scientific areas.

Area	Relation with IP and CV
Artificial Intelligence	Provides models (neural networks, SVM) that interpret CV outputs.
Robotics	Consumes CV data for decision-making (navigation, manipulation).
Machine Learning	Uses descriptors extracted by IP/CV to train classifiers.
Computer Graphics	Inverse path: *model* → *image*; often applies IP for realistic rendering.
Neuroscience	Inspires IP models (e.g., filters similar to retinal ganglion cells).

1.4 PDI Stages

The PDI stages are presented in Figure 1.2, which can be understood as a chain of transformations that reduces data redundancy in search of meaning:

- **Low Level:** Acts directly on the pixels of the noisy image to perform enhancements and filtering, producing as output a clean or enhanced image.
- **Medium Level:** Receives the processed image and performs segmentation and description, transforming the pixel matrix into structured attributes (shape, size, and texture).
- **High Level:** Uses the attribute table to feed logic and artificial intelligence processes, resulting in the final decision or recognition (such as medical diagnosis).

Figure 1.3 details the complete sequence of digital image processing (DIP), from acquisition to interpretation. The flow begins with Image Acquisition (1) and proceeds through Enhancement (2) and Restoration (3). Next, the content is isolated by Segmentation (4) and refined by Morphology (5). The crucial transition occurs in Representation and Description (6), where visual objects are converted into mathematical data (area, perimeter, etc.), enabling Recognition (7). Auxiliary processes include Color Image Processing and Compression, which contribute to the efficiency of storage and analysis.

1.5 Image Formation and the Spectrum

The image formation process is grounded in the interaction between matter and radiant energy. Essentially, an image is conceived when a **sensor records the radiation** resulting from the interaction with a physical object. In the context of human vision and conventional photography, this phenomenon depends on a light source that illuminates the scene; the characteristics of the objects are then encoded through **variations in intensity and color** of the light reaching the sensor, as illustrated in Figure 1.4.

Visible light occupies only a small band of the electromagnetic spectrum — between **380 nm (violet)** and **750 nm (red)** — as illustrated in Figure 1.5. Conventional digital sensors operate within this same window, but specialized equipment can capture radiation invisible to the human eye, such as infrared and X-rays. In digital image processing (DIP), the image formed depends directly on the spectral sensitivity of the sensor used.

From this physical acquisition process, it becomes possible to mathematically model the digital image as a discrete two-dimensional function, in which each point of the scene is represented by numerical samples of light intensity, thus formalizing the concepts of pixel and digital image presented in the next section.

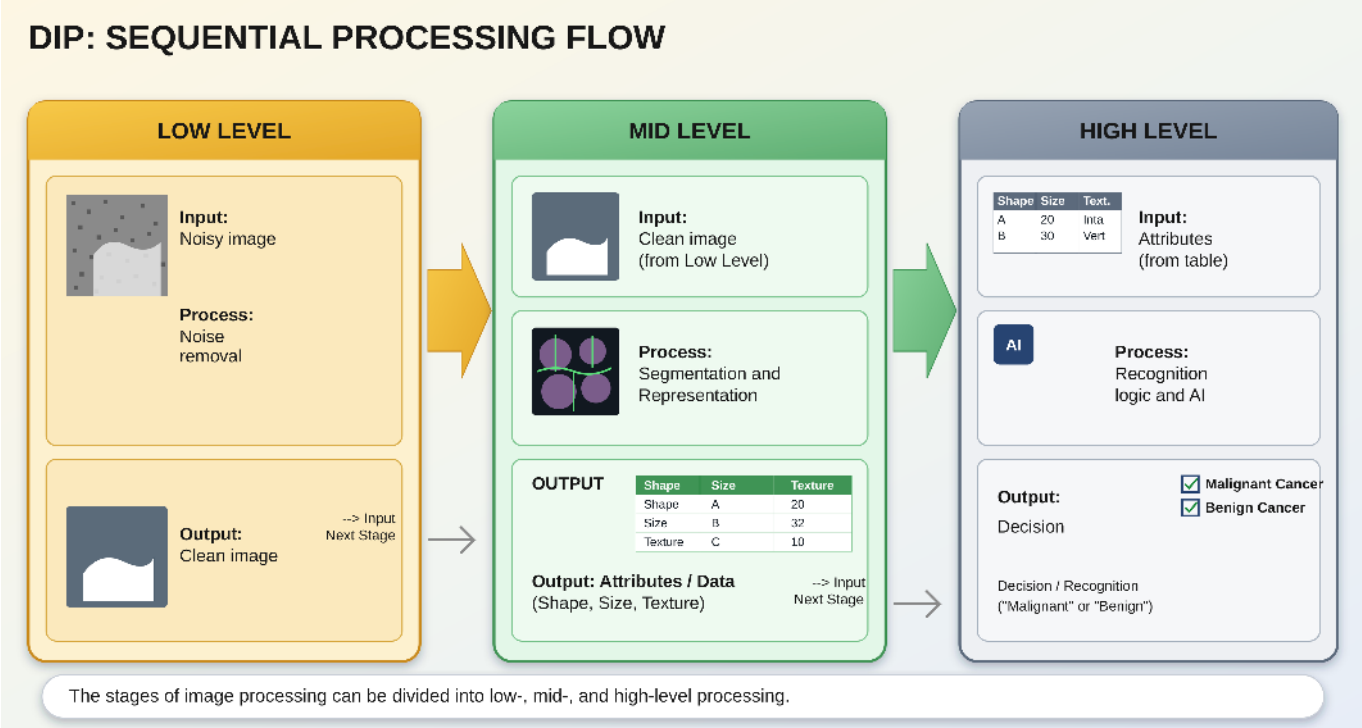


Figure 1.2: Representation of the sequential processing flow: the output of each level becomes the input of the subsequent level.

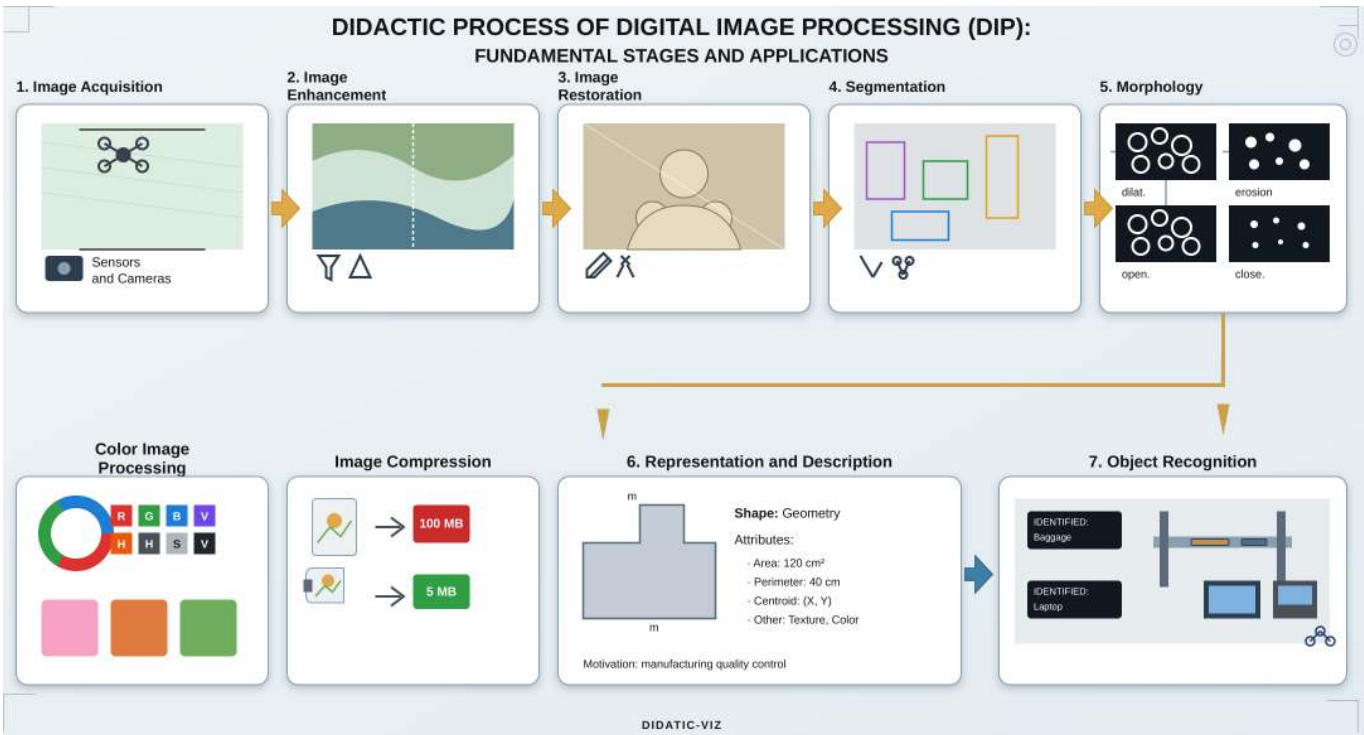


Figure 1.3: Detailed DIP flow: from sensory acquisition to attribute extraction and automated recognition, including color processing and compression.

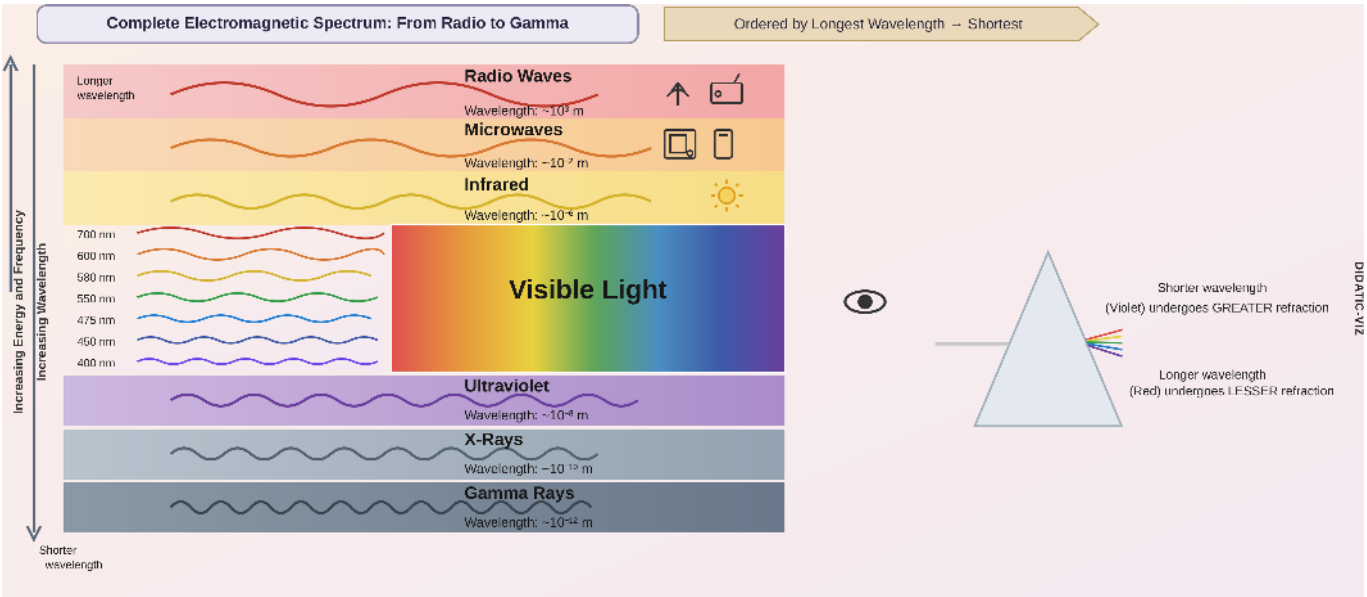


Figure 1.4: Representation of the visible spectrum and its position relative to other electromagnetic radiations, highlighting the variation in wavelengths from 380 nm to 750 nm.

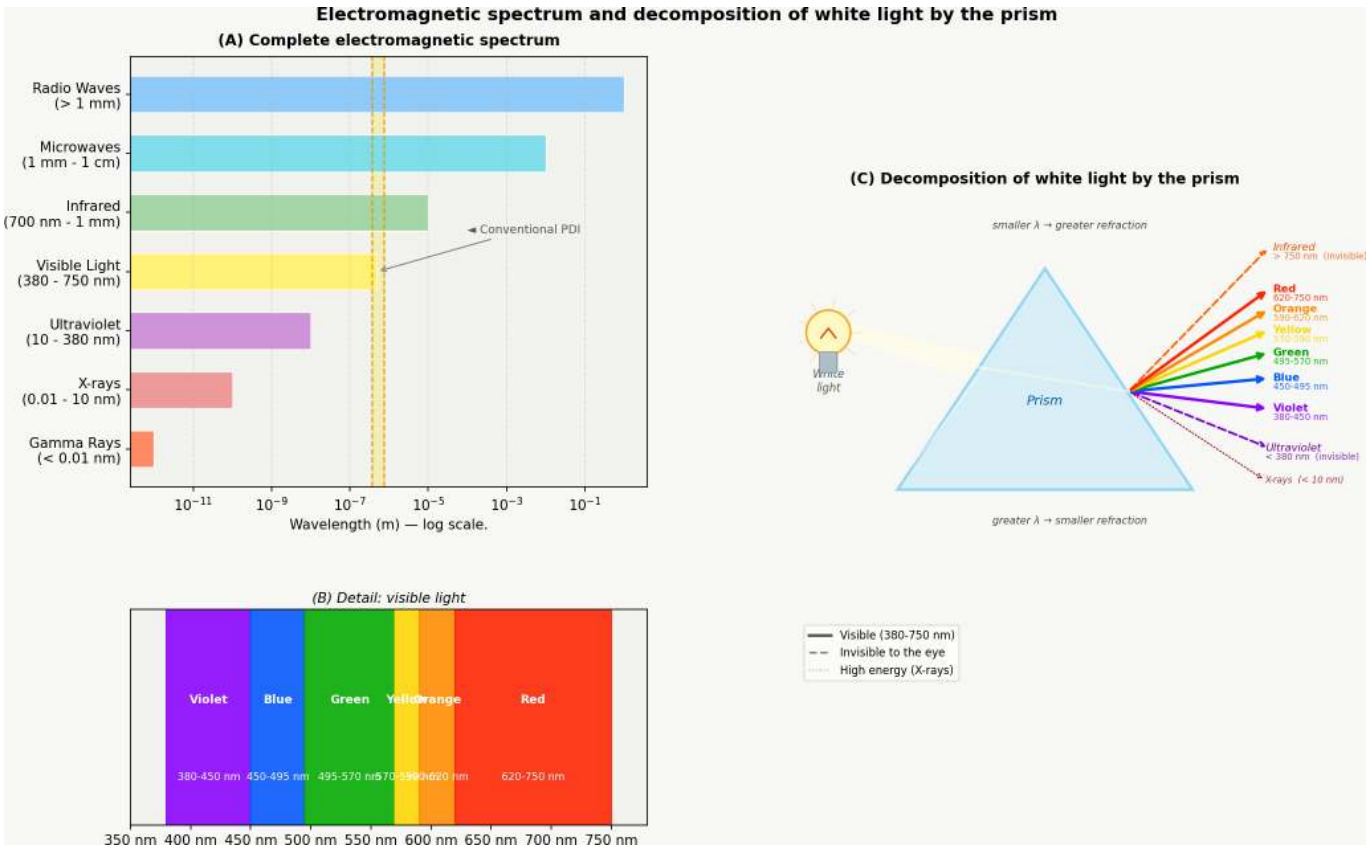


Figure 1.5: (A) Complete electromagnetic spectrum on a logarithmic scale, highlighting the visible range. (B) Detail of visible light (380–750 nm) and its colors. (C) Decomposition of white light by the prism: shorter wavelengths undergo greater refraction, separating UV, visible, and infrared light.

1.6 What is a Digital Image?

A **digital image** is formed by a grid of **pixels** (*Picture Elements*), where each pixel is the smallest elementary unit of the image.

> What is a Pixel?
>
> A **pixel** is the smallest addressable unit that makes up a digital image. Each pixel occupies a unique position in the grid and stores one or more numerical values that represent its intensity or color.

Mathematical Representation

Unlike a continuous function, the domain of a digital image is a finite rectangular plane $\mathbb{E} \subset \mathbb{Z}^2$, which represents the sampling grid. This domain is indexed by integer coordinates:

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

Where:

- L: represents the width of the image (number of columns).
- H: represents the height of the image (number of rows).

The digital image is a function that associates each pair of coordinates (x, y) with one or more values that describe the pixel's appearance.

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

The set $\mathcal{V}$ defines the possible values for the pixel (codomain), varying according to the image type, as demonstrated in Table 1.2.

Table 1.2: Main types of digital images and their respective sets of possible values for each pixel.

Image type	$\mathcal{V}$ (pixel values)	Representation
Binary	$\{0, 1\}$ or $\{0, 255\}$	■□
Grayscale	$\{0, 1, \ldots, 255\}$	`[ ][=][#]`■
Color (RGB)	$\{0, \ldots, 255\}^3$ (ordered triples of values)	■■■

Practical example: A color image in the RGB model can be mathematically represented by a function that associates three intensity values with each pixel. Computationally, this representation corresponds to three overlapping matrices — the red (*Red*), green (*Green*), and blue (*Blue*) channels — in which each element stores the luminous intensity of the respective channel at a given position in the image.

To enable the practical experiments in PDI and VC, this book utilizes the Python scientific ecosystem, integrating libraries focused on matrix manipulation, image processing, and result visualization, as presented below.

1.7 Environment Setup

This material uses Python's scientific ecosystem, with emphasis on **NumPy** (matrix computation), **OpenCV** (computer vision), and the **morph.py** library (ZAMPIROLLI, 2025), developed for didactic purposes and used throughout this book, as presented in Table 1.3.

The project is currently available in **Python** and **Portuguese**. Its organization, however, allows for expansion into other programming languages and languages.

The **Programming Exercises (PEs)** presented at the end of the chapters can be validated by the `testsuite.py` module, which allows running test cases in different languages, including Python, C, C++, Java, JavaScript, and R. Thus, the same test cases can be used to verify solutions developed in the *notebooks* and in **Moodle/VPL**.

Table 1.3: Main libraries and modules used throughout this book for matrix computation, DIP, CV, visualization, and validation of exercises.

Library	Main function
`numpy`	Matrix representation and numerical operations
`opencv-python`	Reading, writing, and computer vision operations
`matplotlib`	Visualization of images and graphs
`morph.py`	Didactic abstraction of DIP operations
`testsuite.py`	Execution and automatic validation of PEs

1.8 Versions of `morph.py`

The `morph.py` library has two public versions:

- **Version 1.0** — original version, published along with the paper presented at EduComp 2024: version 1.0 repository
- **Version 1.1** — version used in the activities of this book: version 1.1 repository

Version 1.1 was adapted to also meet the memory constraints of **Moodle/VPL** (Virtual Programming Laboratory). In version 1.0, some libraries, such as `matplotlib`, `requests`, and `skimage`, were loaded at the time of `import morph`. In environments with restricted resources, this loading could exceed the memory limit and produce errors such as `Jail: out of memory, 128MiB`.

In version 1.1, these *imports* were replaced by **lazy loading**: each library is imported only by the method that actually uses it. Thus, the `import morph` command maintains reduced initial loading, primarily using `numpy` and `cv2`, while additional resources are loaded on demand.

The configuration of the *notebooks* is centralized in the `config.py` file, available in the same repository. This file checks dependencies, seeks to ensure the use of **OpenCV 5.0**, and provides `morph.py` and, when necessary, `testsuite.py`. When the environment does not have the appropriate version of OpenCV, `config.py` can reinstall the required version and, if necessary, request a restart of the execution environment.

Thus, the *notebooks* use a single configuration routine, reducing code repetition and facilitating the reproduction of the computational environment used in this book.

```
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup(testsuite=True)
from morph import mm
from testsuite import TestSuite
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

1.9 Matrix Fundamentals — Caution with Reference Copying

Since a digital image can be represented by a matrix, it is important to understand how to create and manipulate matrices correctly. In Python, there is a common pitfall when using the `*` operator with lists:

> ⚠ Caution with Reference Copying
>
> When executing `m = [[0]*2]*3`, three independent rows are not created. Instead, three references to the **same list** are created. Consequently, modifying an element in one of the rows will also modify the corresponding element in the other rows.

To visualize this behavior, one can run the code in Python Tutor and compare it with the correct way of creating a matrix using lists: `m = [[0]*2 for _ in range(3)]`.

In practice, for digital image processing (DIP), it is recommended to use **NumPy**, which provides efficient multidimensional data structures and vectorized operations suitable for image manipulation. The following code presents different ways of creating synthetic images, whose results are displayed in Figure 1.6.

```python
import numpy as np

# Creating a black image (zeros) of 4, 6 pixels
img_preta = np.zeros((4, 6), dtype='uint8')
img_preta[0,0] = 255  # white pixel in the upper-left corner

# Creating a white image (255) of 4, 6 pixels
img_branca = np.ones((4, 6), dtype='uint8') * 255
img_branca[3,5] = 0  # black pixel in the lower-right corner

# Creating a random image for testing (noise)
img_random = mm.randomImage(4, 6, maxValue=255)

print("Random matrix generated:")
print(mm.drawImg(img_random))

mm.show(
    [img_preta, img_branca, img_random],
    titles=[
        "Predominantly black\n(with 1 white pixel at (0,0))",
        "Predominantly white\n(with 1 black pixel at (3,5))",
        "Random image\n(noise simulation)"
    ],
    cols=3,
    figsize=(9, 3),
    axis=True
)
```

```
Random matrix generated:
160  64  97  26  40 172
171 167 233  81  97 217
217 183  31 200  94 143
118  25  48  56 190 160
```

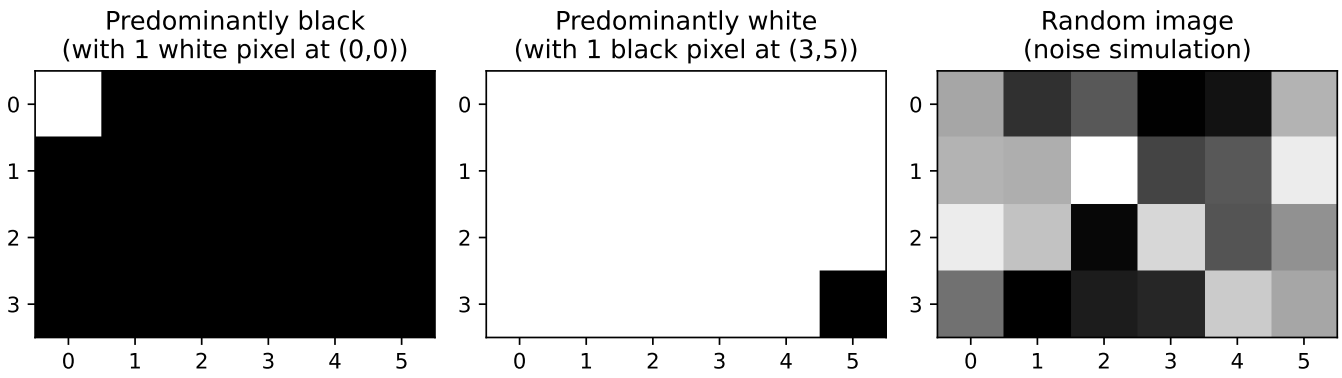


Figure 1.6: Examples of synthetic images represented as matrices.

1.10 Reading and Displaying Images

In libraries such as NumPy, OpenCV, and scikit-image, digital images are often computationally represented as NumPy *arrays*. In this way, matrix operations and linear algebra concepts can be applied directly to digital image processing (DIP).

One of the fundamental operations in digital image processing (PDI) is image reading.

In the `morph.py` library, the function `mm.read()` allows loading images from both local files and URLs, as illustrated in Figure 2.7..

```
import os
import numpy as np

url     =
"https://upload.wikimedia.org/wikipedia/commons/8/87/Mandrillus_sphinx_339428057.jpg"
caminho = "imagens/mandrill.png"

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"Dimensions (H, W, Channels): {img.shape}")
print(f"Data type: {img.dtype}")

mm.show(img, title="Example: RGB capture")
```

```
Dimensions (H, W, Channels): (1365, 2048, 3)
Data type: uint8
```

Example: RGB capture

Figure 1.7: Mandrill (Mandrillus sphinx) in natural environment. Credit: Julien Renoult (CC BY 4.0).

1.10.1 Alternative: *Downloading* the Image for Local Storage

In environments where direct reading from URLs is unavailable—due to network restrictions, firewall policies, or lack of connectivity—an alternative is to download the image to the local file system beforehand. Once stored, the image can be loaded normally using the `mm.read()` function. In the example below, the file is saved locally with the name `mandrill.png`.

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

```
img = mm.read('mandrill.png')
mm.show(img, title="Example: RGB Capture (local file)")
```

1.10.1.1 Explanation

- `wget -O mandrill.png <URL>` downloads the image and stores it locally with the specified name;
- `mm.read('mandrill.png')` reads the file directly from the file system, without requiring additional HTTP requests;
- this strategy reduces dependence on connectivity during experiments and avoids repeated downloads of the same image.

Note

The `!` prefix is used in notebook-based environments, such as Jupyter Notebook, JupyterLab, and Google Colab, to execute operating system commands directly in code cells. In conventional terminals, the command must be used without the `!` prefix.

If the `wget` utility is not installed, the following alternative can be used:

```
!curl -o mandrill.png \

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

1.11 Type Conversion and Thresholding

As we have seen, a color image in the RGB space is represented by the function:

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

That is, for each pixel (x, y), we have three values (R, G, B) that define its color.

Conversion to Grayscale

To convert an RGB image to grayscale, it is necessary to combine the three channels into a single intensity value g, which represents the perceived brightness. Since the human eye is not equally sensitive to red, green, and blue, a **weighted average** is used. The ITU-R BT.601 standard ({ITU-R}, 2011) defines the following weights:

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

After the calculation, the value g is rounded to the nearest integer and adjusted to the interval $[0, 255]$. The result is a new image, now in grayscale, represented by:

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

Thresholding

From the grayscale image $f_{\text{gray}}(x, y)$, a fundamental operation is **thresholding**, which produces a **binary** image (only black and white). To do this, a cutoff value T is chosen (usually in the interval $[0, 255]$) and defined as:

$$f_{\text{bin}}(x, y) = \begin{cases} 255 & \text{if } f_{\text{gray}}(x, y) > T \\ 0 & \text{otherwise} \end{cases} \tag{1.4}$$

Example: With $T = 128$, pixels with intensity above 128 become white (255); the remaining ones become black (0).

Thresholding is widely used to **segment objects from the background**, extract edges, or create binary masks for further processing.

> **Note:** The value 255 represents maximum white in 8-bit images, while 0 represents absolute black.

Practical example of conversion and thresholding

Figure 1.8 illustrates the main steps to transform a color image into grayscale and then convert it into a binary image by thresholding. The following code implements these steps:

```
# 1. Convert to Grayscale
img_gray = mm.gray(img)

# 2. Apply threshold (Pixels > 128 become 255, others 0)
limiar = 128
img_binaria = mm.threshold(img_gray, limiar)

# Using the new function
mm.show(
    [img, img_gray, img_binaria],
```

```
    titles=["Original", "Grayscale", f"Binary (T={limiar})"],
    cols=3
)
```

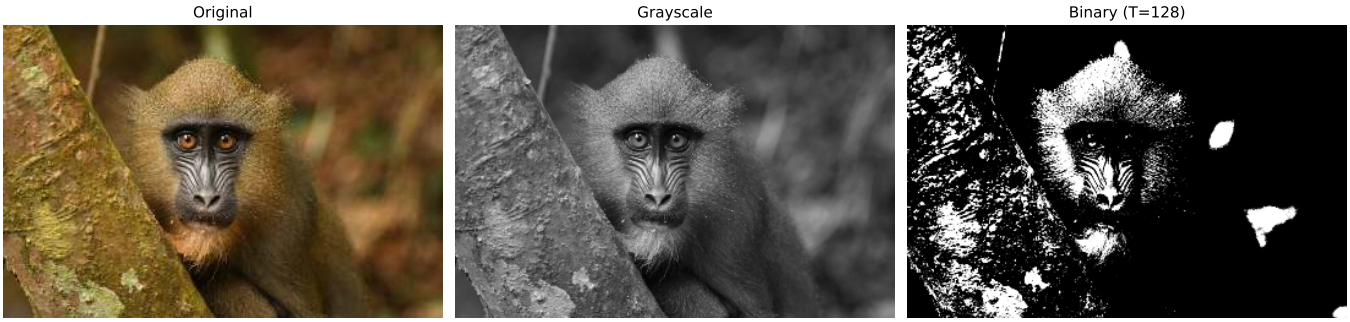


Figure 1.8: Basic image processing: (a) original image, (b) grayscale image, (c) image binarized by threshold (T=128).

1.12 Otsu's Method Thresholding

As presented in Equation 1.4, thresholding converts a grayscale image into a binary one using a cutoff value T. So far, we have manually set $T = 128$.

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

However, manually choosing T is not always trivial. The `mm` library offers an automatic alternative: when the `limiar` parameter is **not provided**, the `mm.threshold(img_gray)` function computes the value of T using **Otsu's method** (OTSU, 1979). This method, which will be detailed in future chapters, maximizes the inter-class variance of the histogram (frequency of each gray level), automatically separating object and background pixels.

The code below compares manual thresholding ($T = 128$) with automatic thresholding (Otsu), also showing the computed value of T:

```
import cv2
# Thresholding with fixed T (manual)
T_fixo = 128
img_bin_fixo = mm.threshold(img_gray, T_fixo)

# Thresholding by Otsu's method (automatic T)
T_otsu, img_bin_otsu = cv2.threshold(img_gray, 0, 255,
                                     cv2.THRESH_BINARY + cv2.THRESH_OTSU)
print(f'Threshold calculated by Otsu: T = {T_otsu}')
# or simply:
# img_bin_otsu = mm.threshold(img_gray)

# Side-by-side display
mm.show(
    [img_gray, img_bin_fixo, img_bin_otsu],
    titles=["Grayscale", f"Binary (T={T_fixo})", f"Binary (Otsu, T={T_otsu})"],
    cols=3
)
```

```
Threshold calculated by Otsu: T = 95.0
```

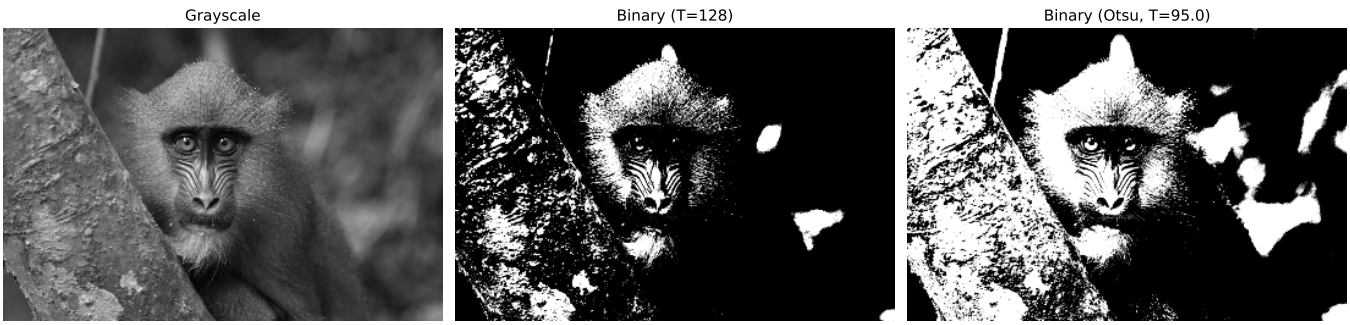


Figure 1.9: Comparison between manual thresholding (T=128) and automatic (Otsu) on the grayscale image.

The Figure 1.9 shows that the threshold obtained by Otsu automatically adapts to the image, resulting in more efficient binarization than a fixed value, especially when the object and background intensities are well separated in the histogram. This technique is widely used in CV systems for document binarization, object detection, and image preprocessing.

💡 **Simplicity of the `morph.py` library**

While OpenCV requires the full call:

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

the `mm` library abstracts all this complexity: simply call `mm.threshold(img_gray)`. The Otsu threshold is calculated automatically and the binary image is returned directly. This approach allows one to focus on the concept rather than implementation details.

1.13 Accessing Pixels

In Python with NumPy, an image is structured as a **multidimensional array**. Accessing a specific pixel uses the matrix convention of **row** (Y axis) and **column** (X axis): `img[row, column]`.

The code in Figure 1.10 demonstrates how to extract these values in color (RGB) and grayscale images, as well as how to isolate the immediate neighborhood of the point of interest.

```
import matplotlib.pyplot as plt

# 1. Definition of the target pixel coordinates
r, c = 600, 800

# 2. Direct access to pixel values
pixel_cinza = img_gray[r, c]
pixel_rgb   = img[r, c]

print(f"  VALUES AT THE TARGET PIXEL ({r}, {c}):")
print(f"   • Grayscale (Scalar)       : {pixel_cinza}")
print(f"   • Colored (RGB Vector)     : R={pixel_rgb[0]}, G={pixel_rgb[1]}, B={pixel_rgb[2]}")
print("-" * 50)

# 3. Didactic display of the 3x3 neighborhood around the pixel (r, c)
# Crops from row r-1 to r+1, and from column c-1 to c+1
vizinhanca_gray = img_gray[r-1:r+2, c-1:c+2]

print(f"  3x3 NEIGHBORHOOD MATRIX IN GRAYSCALE:")
print(f"   (The central target pixel is highlighted with brackets)\n")

# Formatted printing to highlight the central pixel
```

```python
for i, linha in enumerate(vizinhanca_gray):
    linhas_str = []
    for j, valor in enumerate(linha):
        if i == 1 and j == 1:
            linhas_str.append(f"[{valor:3d}]") # Highlights the center (100, 100)
        else:
            linhas_str.append(f" {valor:3d} ")
    print("   " + " ".join(linhas_str))

# 4. Visual demonstration using mm.show (optional, but excellent for the book)
# Shows the entire image with a point marking the region
plt.figure(figsize=(5, 5))
plt.imshow(img)
plt.plot(c, r, 'ro', markersize=8, label=f'Pixel ({r},{c})') # Plots (x, y) -> (c, r)
plt.title(f"Localização do Pixel ({r}, {c}) na Imagem")
plt.legend()
plt.axis('on') # Keeps the axes so the student can see the coordinates 100, 100
plt.show()
```

```
 VALUES AT THE TARGET PIXEL (600, 800):
  • Grayscale (Scalar)       : 81
  • Colored (RGB Vector)     : R=92, G=78, B=67
----------------------------------------------------
 3x3 NEIGHBORHOOD MATRIX IN GRAYSCALE:
  (The central target pixel is highlighted with brackets)

    83    96   105
    85  [ 81]   96
    83    77    84
```

Figure 1.10: Accessing a specific pixel (r, c) and the representation of its neighborhood in the image.

Line-by-line explanation:

1. `img_gray[r, c]` — Returns a single integer value (scalar) between 0 and 255, representing the brightness intensity of the gray pixel.
2. `img[r, c]` — Returns a vector with three values `[R, G, B]`, corresponding to the intensities of the Red, Green, and Blue channels.

3. `img_gray[r-1:r+2, c-1:c+2]` — Performs slicing to extract the 3×3 submatrix around the pixel. This operation is the basis for implementing spatial filters and convolutions.
4. The subsequent `for` loop merely formats this submatrix in the console, displaying the central pixel within square brackets `[ ]` for didactic purposes.
5. The function `plt.plot(c, r, 'ro')` plots a red point over the image to correlate the numerical coordinate with its visual position. Note that Matplotlib uses the standard screen order `(X, Y)`, inverting it to `(c, r)`.

Since indexing in Python is zero-based, the top-left corner of the image is the coordinate `(0,0)`. Always keep this rule in mind: the first dimension of the matrix controls the **height** (rows/Y) and the second controls the **width** (columns/X).

1.14 Summary

In this chapter, the fundamentals of digital image representation were presented: the definition of pixel, the structuring of images into matrices, and the impact of sampling and quantization on final quality:

- **Digital image** = function $f(x, y)$ that maps coordinates to intensities (scalar or vectorial).
- **Domain**: finite set $\mathbb{E} = \{(x, y) \in \mathbb{Z}^2 \mid 0 \leq x < L,\ 0 \leq y < H\}$.
- **Main types**: binary ($\mathcal{V} = \{0, 255\}$), grayscale ($\mathcal{V} = [0, 255]$), and RGB ($\mathcal{V} = [0, 255]^3$).
- **Thresholding** converts grayscale into binary; **Otsu's method** automatically determines the cutoff value by maximizing between-class variance.
- The `morph.py` library (or `mm`) provides educational functions for basic digital image processing (DIP) operations, such as `mm.gray()`, `mm.threshold()`, and `mm.show_multiple()`.
- **NumPy Trap**: never use `[[0]*n]*m` to create matrices — always use `np.zeros()` or `np.ones()`.
- Pixel access via `img[row, column]`, with ***zero-based*** indexing.

Chapter 2 will cover **histograms and contrast equalization**.

1.15 🤖 Using Gemini Notebook as a Complementary Tutor

In this edition, in addition to the interactive *notebooks* on Google Colab, **Gemini Notebook** is available as a complementary study tool. The platform uses exclusively the documents provided by the author as its knowledge base, ensuring responses that are consistent with the book's content.

🎓 Study with the Intelligent Tutor

Access the chapter's environment via the *link* below and especially explore the **Study Guide** and **Conversation** options to deepen your understanding.
🚀 ACCESS Gemini Notebook: CHAPTER 01

🌐 **Language and Programming Language**

The project for this chapter in Gemini Notebook was built using only the **Portuguese** text and the **Python** code examples. If you are studying from the English or French edition, or following the C++ track, the tutor's responses may not exactly correspond to the version you are reading.

⚠️ **Notice about AI-Generated Content**

AI is a powerful study ally, but the generated content may contain **errors or inaccuracies**. Always consult **books, scientific articles, and other reliable academic sources** to validate the information. Whenever possible, run the practical examples provided in this chapter to verify the results.

Available Platform Features

The **Gemini Notebook** offers an advanced suite of AI-based tools to transform the static content of the book into a dynamic, multimedia learning experience. The platform employs **RAG** (*Retrieval-Augmented Generation*) techniques, grounded in the work of Lewis (2020), to base responses strictly on the provided documents and minimize the occurrence of hallucinations.

The main features include:

- **Multimodal Summaries (Audio and Video)**: Generation of natural conversations between experts in the **Audio Summary** format (*podcast*-style) and **Video Summary**, discussing the central themes of the chapter, such as the differences between PDI and VC, or the interpretation of transformations like thresholding and Otsu's method.
- **Structure Visualization (Mind Map and Infographic)**: Automatic creation of diagrams that visually connect concepts, for example, the processing flow from digital image capture, through conversion to grayscale, thresholding, and binary segmentation.
- **Assessment Tools (Quizzes and Flashcards)**: Generation of multiple-choice **Quizzes** and **Flashcards** for knowledge reinforcement, based on the authorial text (e.g., questions about the RGB-to-grayscale conversion formula or about the operation of global and Otsu thresholds).
- **Presentation Support (Slides and Reports)**: Assistance in structuring **Slide Presentations** and in writing technical **Reports**, facilitating the communication of experimental results with images.
- **Data Analysis (Data Table)**: Organization of data extracted from the text into structured tables, aiding the understanding of practical examples, such as the comparison between different threshold values.
- ***Contextual Chat***: Enables direct questioning about the code and theory, such as: *"How can I implement RGB-to-grayscale conversion using the weights of the ITU-R BT.601 standard?"* or *"What happens to the binary image if I choose a threshold T=200 instead of T=128?"*.

1.16 Exercise List

1. (15%) In your own words, define **digital image** and **pixel**. Give a concrete example of how a color (RGB) image is represented in matrix form in the computer.

2. (15%) Explain the differences between a **binary image**, **grayscale (8-bit)**, and **RGB color** image, indicating the range of possible values for each pixel in each type.

3. (20%) Considering the RGB → grayscale conversion formula from the ITU-R BT.601 standard:

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

   Compute the grayscale pixel value for $(R, G, B) = (80, 180, 30)$. Round to the nearest integer.

4. (20%) What is **thresholding**? Explain the difference between choosing a fixed threshold T (e.g., $T = 128$) and using the **Otsu method** for automatic threshold determination. In a few words, how does the Otsu method select the threshold?

5. (15%) In the context of the didactic library `mm` discussed in the chapter, answer:
   - a) (7.5%) How do you access the pixel value at position (row=50, column=60) of a grayscale image `img_gray`?
   - b) (7.5%) What is the advantage of using `mm.threshold(img_gray)` without passing the threshold? Compare it with the equivalent call in OpenCV.

6. (15%) What does the `img.shape` property return for a NumPy image in RGB format? Provide a concrete example with a 640×480 pixel image.

1.17 Chapter References

The theoretical foundation of this chapter comprises the following works on DIP and CV:

- Gonzalez (2018) for the fundamentals of **Digital Image Processing** (DIP).
- Singh (2019) for the practical implementation of **image processing and analysis** methods.
- Szeliski (2022) for the study of CV and fundamental algorithms.
- Bradski (2008) for the application of the **OpenCV** library in a Python environment.
- Lewis (2020}) for the concept of **retrieval-augmented generation (RAG)**, used to support the processing of information in this material.

Run Colab Open GitHub

This appears to be a Markdown document containing only image links with badges. As there is no Portuguese prose text to translate, the content remains unchanged, preserving all Markdown structure and URLs exactly as provided.

1.18 💻 Practical Part with Programming Exercises

1.18.1 🎯 Objective of this Notebook

The **Programming Exercises (PEs)** presented below can also be submitted in Moodle activities (VPL activities) that provide automatic *feedback*.

This notebook was developed to overcome limitations of Moodle usage. With it, you should:

1. **Develop:** Write and edit your solution directly in the Colab environment.
2. **Validate:** Test your code locally using the **same test cases** as those in Moodle.
3. **Organize:** Save your codes from the VPL activities securely.
4. **Evaluate:** When connected to Moodle, simply copy your solution and click on **Evaluate** in Moodle (**if you are on the UFABC network**) to record your official grade.

1.18.2 § Step-by-Step Instructions

In an execution environment (such as VSCode, Jupyter, or Colab), follow the order below to set up the environment and validate your exercises:

1.18.2.1 Environment Preparation

Run the code cell below to download `morph.py` and `testsuite.py` from the course repository — only if they do not already exist in the local directory. With both files in `./`, the *notebook* and the `TestSuite` subprocesses find the module without any extra path configurations.

Note: The `testsuite.py` script will automatically look for test cases in `all/{cap}/cases` on GitHub.

1.18.2.2 Writing the Code

Save your solution in a code cell using the magic command `%%writefile`. The file name must follow the pattern `EPX_Y.*`, where `X` is the chapter, `Y` is the exercise, and `*` is the language extension.

> **Example:** `%%writefile EP01_01.py`

1.18.2.3 *Download*

Download `morph.py` and `testsuite.py` by running the cell below:

```
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")
```

```
import config
config.setup(testsuite=True)
from morph import mm
from testsuite import TestSuite
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

1.18.2.4 Running the Tests

After saving the file with your solution, run the command below (in a new cell) to evaluate the automated tests:

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

Replace the extension according to the language used:

Language	Extension
Python	`.py`
Java	`.java`
C	`.c`
C++	`.cpp`
JavaScript	`.js`
R	`.r`

> **How it works:** The `TestSuite` downloads the test cases from GitHub, runs your program with each input, and compares the output with the expected one – automatically calculating your grade.

To test Python code directly, without saving a file, use `run_code(codigo)` passing the code as a *string* in a variable `codigo`:

```
codigo = """
from morph import mm
# ... your code here ...
"""
TestSuite("EP04_01").run_code(codigo)
```

1.18.3 ⚠️ Important: Rules and Best Practices

1.18.3.1 🔹 About Data Input

Your program must read from standard input (keyboard).

- **Python:** Use `input()`.
- **Other languages:** Use the equivalent standard reading command (`cin`, `Scanner`, etc.).

1.18.3.2 🔹 AI Configuration in Colab

For better learning, it is recommended to disable AI code autocomplete, as it will not be available during evaluations. For example, in the Chrome browser:

- Go to: **Tools > Settings > Generative AI**
- Uncheck: **Enable code generation**

1.18.3.3 🔹 Academic Integrity (Plagiarism)

This local testing feature applies to EPs **without variations**. However:

- **Individuality:** Each student must develop their own solution.
- **Similarity Detection:** The instructor uses tools that detect copies, **even with changes to variable names or whitespace**.

1.18.4 EP01_01 📏 Three distance metrics in DIP

In this activity, you must write a program that computes the three classical distances in DIP: **Euclidean (L2)**, **City-Block (L1)**, and ***Chessboard* (L∞)**.

- Read **4 real numbers** representing the coordinates: A_x, A_y, B_x, B_y.
- Compute the three distances using the formulas:

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

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

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

- Print the three results, each on a separate line, formatted with **two decimal places**, in the following order: **Euclidean**, ***City-block***, ***Chessboard***.

📌 **Important**:

- Use the standard mathematical functions of your language: `math.sqrt`, `abs` (or `fabs`), and `max`.
- The output must contain **only the numbers** (one per line), without additional text.
- See an interactive simulator for this problem at the **Figure 1.11** (graph with draggable points and visualization of the three metrics).

1.18.4.1 🖼️ Why does this matter? – Computational cost

In a **1000×1000 pixel** image (1 million pixels), computing the distance from each pixel to a reference point requires **1 million operations**. The choice of metric affects performance:

Metric	Operations per pixel	Relative cost (1M pixels)	When to use
Euclidean (L2)	2 subtractions, 2 multiplications, 1 addition, 1 `sqrt`	🔴 Most expensive – `sqrt` is costly	"Real" distance in continuous space
***City-block* (L1)**	2 subtractions, 2 `abs`, 1 addition	🟡 Moderate – no square root	Grids, robotics, binary images
***Chessboard* (L∞)**	2 subtractions, 2 `abs`, 1 `max`	🟢 Most efficient	Piece movements, morphology

The `sqrt` function is computationally more expensive than operations such as addition, subtraction, multiplication, and absolute value. On modern CPUs, the difference can be small (about 1.5× to 3×), but in embedded systems or in loops with millions of iterations, any gain matters. Therefore, **when the goal is only to compare distances** (e.g., finding the nearest point), use the **squared Euclidean distance**.

1.18.4.2 📋 Task (specification for VPL)

Input:
A single line with four real numbers: `Ax Ay Bx By`

Output:
Three lines, each with a real number with two decimal places (Euclidean, City-block, Chessboard).

1.18.4.3 📌 Examples

Input	Output	Observation
0	5.00	3-4-5 triangle
0	7.00	
3	4.00	
4		
0	1.41	Unit diagonal
0	2.00	
1	1.00	
1		

Example of testing `sqrt` in Python, with `timeit` isolating each operation:

```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"Simple sum          : {t_soma:.3f} s")
print(f"Sum + sqrt          : {t_sqrt:.3f} s")
print(f"Ratio (sqrt/sum)    : {t_sqrt/t_soma:.2f}x")
```

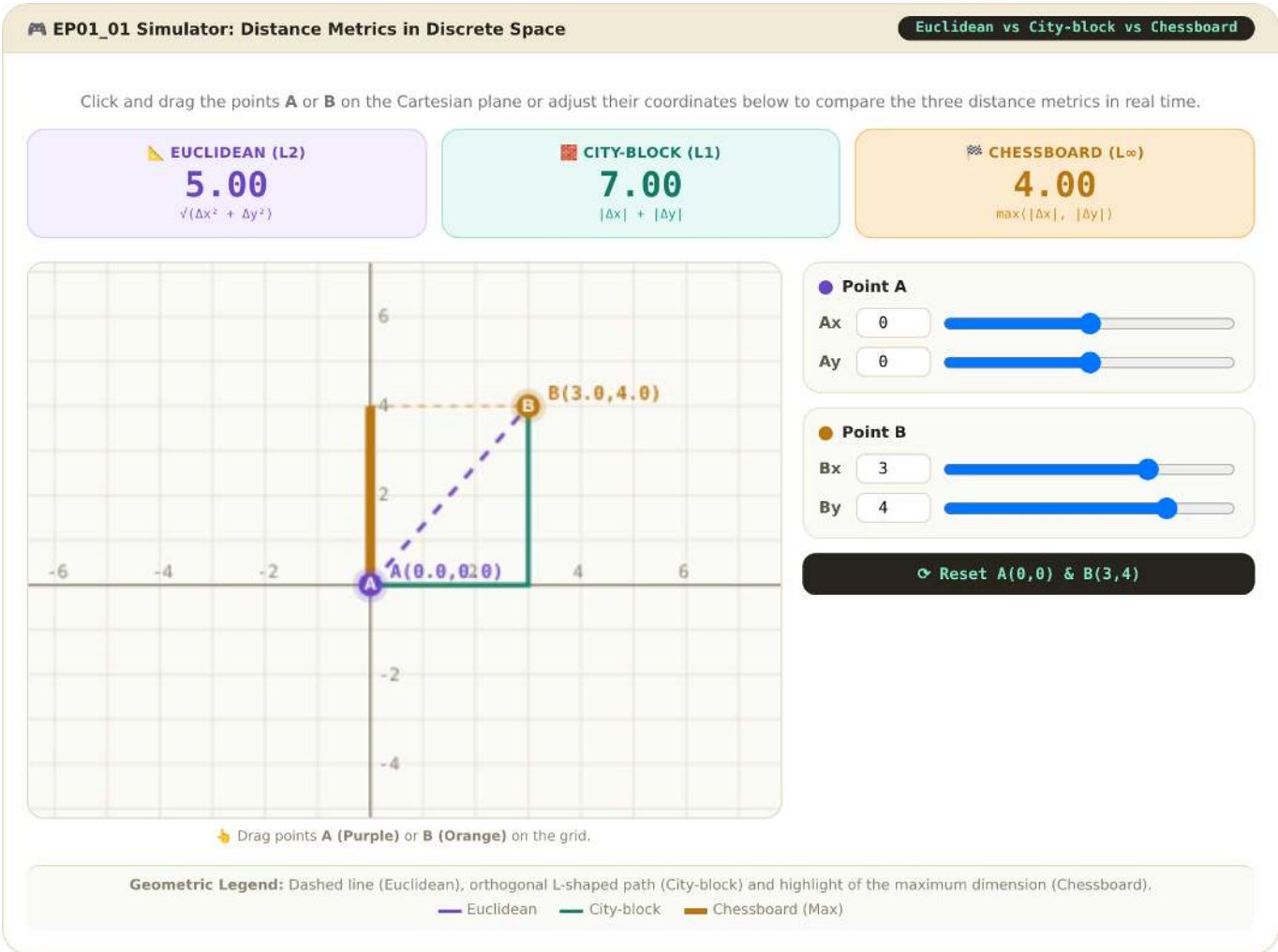
```
Simple sum          : 2.734 s
Sum + sqrt          : 5.744 s
Ratio (sqrt/sum)    : 2.10x
```

1.18.4.4 🐍 Python

Simply create a regular code cell and insert the Python code. Input can be simulated using `input()`, which works as usual.

Example cell:

```python
%%writefile EP01_01.py
# Python code
x1,y1,x2,y2 = int(input()), int(input()), int(input()), int(input())
# Calculation of differences
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap01/sim-ep0101-distancia.html

Figure 1.11: EP01_01 Simulator: Euclidean, City-block, and Chessboard Distances

```python
dx = abs(x2 - x1)
dy = abs(y2 - y1)

# 1. Euclidean distance (L2)
dist_euclidiana = (dx**2 + dy**2)**0.5

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

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

# Output formatted according to the test cases
print(f"{dist_euclidiana:.2f}")
print(f"{dist_city_block:.2f}")
print(f"{dist_chessboard:.2f}")
```

```
Overwriting EP01_01.py
```

```python
# Expects you to type 4 integers when running this cell.
# In Jupyter or Google Colab, the %run -i magic allows the script to read from the keyboard.
# In a regular terminal, you would use: python3 EP01_01.py (without the '!' and '%run').

# %run -i EP01_01.py
```

```python
# Sends 4 integers as standard input (stdin) to the script EP01_01.py using a pipe
!echo -e "0\n0\n4\n4" | python3 EP01_01.py
```

```
5.66
8.00
4.00
```

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

```
<IPython.core.display.HTML object>
```

1.18.4.5 ☕ Java

To run Java in Colab, you need to use a cell with the `%%writefile` prefix to save the code to a file, compile, and run it.

```java
%%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));
    }
```

```
}
```

```
Overwriting 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()
```

```
<IPython.core.display.HTML object>
```

1.18.4.6 💻 C

Similarly, use `%%writefile` to save the code, then compile and run. For C, we use the **GCC** compiler.

```
# Install GCC compiler and build tools
# build-essential includes gcc, g++, make, etc.
import platform, shutil, subprocess

def instalar_gcc():
    if shutil.which("gcc"):
        print("  GCC already available."); return
    if platform.system() != "Linux":
        print("  Mac: xcode-select --install | Windows: WSL or MinGW"); return
    try: import google.colab; cmd = ["apt-get", "install", "-y", "build-essential"]
    except ImportError:       cmd = ["sudo", "apt-get", "install", "-y", "build-essential"]
    subprocess.run(cmd, check=True)
    print("  build-essential installed!")

instalar_gcc()
```

```
  GCC already available.
```

```
%%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;
}
```

```
Overwriting EP01_01.c
```

```
# Compiles the .c file generating the executable EP01_01
# -lm is used to link the math library (math.h) if necessary

!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()
```

```
<IPython.core.display.HTML object>
```

1.18.4.7 💻 C++

Similar to Java, use `%%writefile` to save the code, then compile and run it. Remember that, in Colab, you also need to install the following:

```
# Install the G++ compiler for C++
# build-essential includes g++, make, etc.
import platform, shutil, subprocess

def instalar_gpp():
    if shutil.which("g++"):
        print("  G++ already available."); return
    if platform.system() != "Linux":
        if platform.system() == "Darwin":
            print("  Mac: xcode-select --install")
        else:
            print("  Windows: use WSL or MinGW (https://www.mingw-w64.org)")
        return
    try: import google.colab; cmd = ["apt-get", "install", "-y", "build-essential"]
    except ImportError:       cmd = ["sudo", "apt-get", "install", "-y", "build-essential"]
    subprocess.run(cmd, check=True)
    print("  C++ compiler ready.")

instalar_gpp()
```

```
  G++ already available.
```

```
%%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;
}
```

```
Overwriting 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()
```

```
<IPython.core.display.HTML object>
```

1.18.4.8 🌐 JavaScript (Node.js)

For JavaScript, use `%%writefile` to create the file and run it with 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));
});
```

```
Overwriting EP01_01.js
```

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

```
<IPython.core.display.HTML object>
```

1.18.4.9 📊 R

In Colab, R code can be run directly using the `%%R` magic command.
The program should read **four numbers** (x1, y1, x2, y2) and display the Euclidean distance with **two decimal places**.

Example cell:

```
%%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))
```

```
# Install R and Rscript
# r-base installs the complete R environment, including Rscript
import platform, shutil, subprocess

def instalar_r():
    if shutil.which("R"):
        print("  R is already available."); return
    if platform.system() != "Linux":
        if platform.system() == "Darwin":
            print("  Mac: https://cran.r-project.org/bin/macosx/")
        else:
            print("  Windows: https://cran.r-project.org/bin/windows/base/")
        return
    try: import google.colab; cmd = ["apt-get", "install", "-y", "r-base"]
    except ImportError:       cmd = ["sudo", "apt-get", "install", "-y", "r-base"]
    subprocess.run(cmd, check=True)
    print("  R environment ready.")

instalar_r()
```

```
  R is already available.
```

To test in the same way as in the previous examples, standard input (stdin) should be used in the terminal or the code should be adapted as follows:

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

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

```
Overwriting EP01_01.r
```

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

```
5.66
8.00
4.00
```

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

```
<IPython.core.display.HTML object>
```

1.18.5 EP01_02 📊 Predictive Performance — ML Metrics in CV

In this activity, you will dive into the world of **Machine Learning**. Your goal is to evaluate the performance of a binary classifier by calculating metrics from a **Confusion Matrix**.

1.18.5.1 🧠 Why does the right metric matter?

Imagine a **R$ 1.00 coin** detector. The impact of the error defines the priority metric:

Metric	Practical Example	Importance in IP/CV
Accuracy	**Grain Counting**	Useful when classes are balanced (e.g., half of the grains defective, half healthy).
Precision	**Security/Biometrics**	Crucial to avoid **False Positives** (e.g., not allowing an impostor to access a system due to recognition errors).
Sensitivity	**Health (Tumors)**	Crucial to avoid **False Negatives** (e.g., not letting a tumor go unnoticed in an X-ray examination).
F1-score	**Banknotes**	Ideal for a **balance** between not rejecting genuine notes and not accepting counterfeit ones.

1.18.5.2 📊 The Confusion Matrix

	Predicted Positive	**Predicted Negative**
Actual Positive	**TP** (True Positive)	**FN** (False Negative)
Actual Negative	**FP** (False Positive)	**TN** (True Negative)

Task:

1. Read **4 integer values** in the order: `TP`, `FN`, `FP`, `TN`.
2. Calculate the metrics using the formulas:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Sensitivity (Recall)} = \frac{TP}{TP + FN}$$

$$\text{F1-score} = \frac{2 \times \text{Precision} \times \text{Sensitivity}}{\text{Precision} + \text{Sensitivity}}$$

3. Print the results formatted with **two decimal places**, one per line.

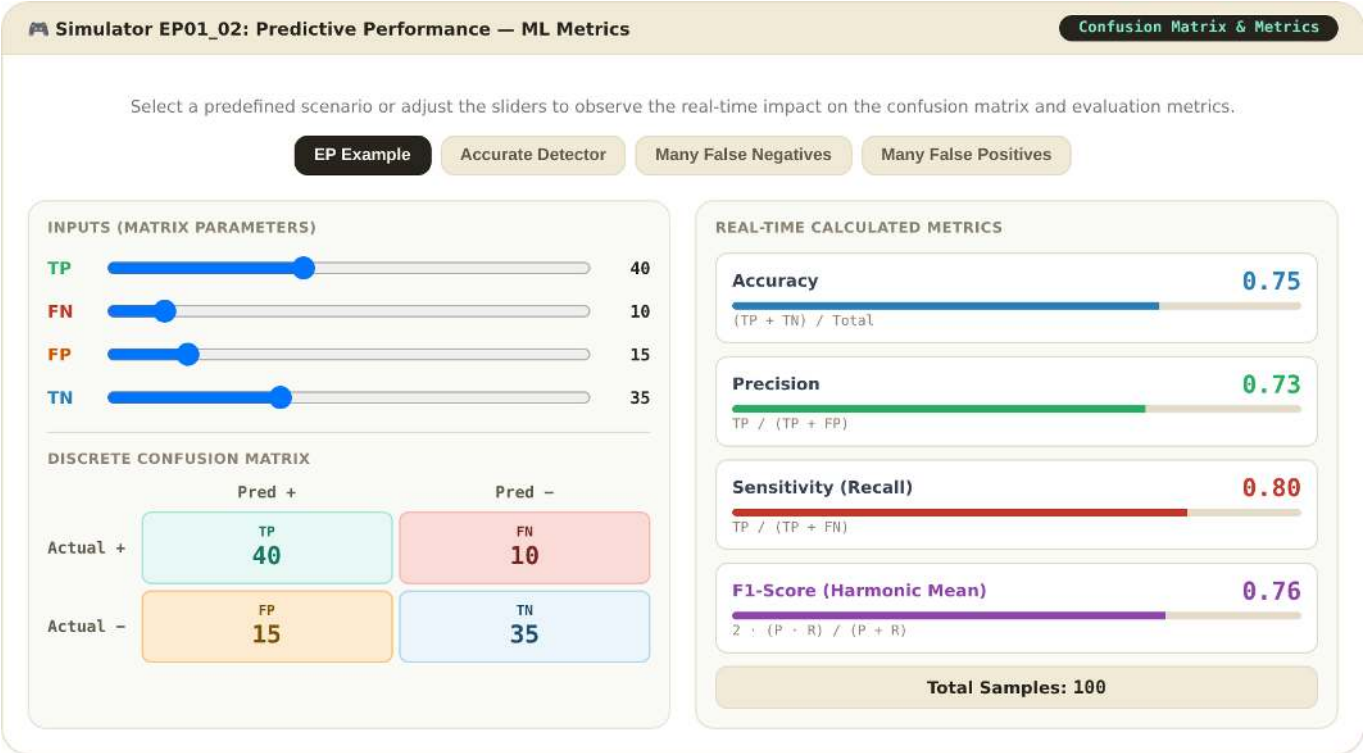
📌 **Important**:

- Use floating-point division to avoid truncated results.
- The output order must be: **Accuracy**, **Precision**, **Sensitivity**, and **F1-score**.
- See the interactive simulation for this EP at Figure 1.12.

1.18.5.3 📌 Example Execution

Input	Output	Observation
40	0.75	Accuracy
10	0.73	Precision
15	0.80	Sensitivity
35	0.76	F1-score

(Note: TP=40, FN=10, FP=15, TN=35. Total cases = 100)



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap01/sim-ep0102.html

Figure 1.12: EP01_02 Simulator: Predictive Performance — ML Metrics

```
# your solution
```

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

```
<IPython.core.display.HTML object>
```

1.18.6 EP01_03 ↗ *Mean Average Precision* (mAP) — Precision-Recall Curve

In this activity, you will evaluate a binary classifier (e.g., deforestation detection in satellite images, see dgi.inpe.br) using the **Precision-Recall curve** and the **mAP (*Mean Average Precision*)** metric. mAP is standard in competitions such as COCO (*Common Objects in Context*) and PASCAL VOC (*Visual Object Classes*) and in YOLO (*You Only Look Once*) models.

1.18.6.1 🧠 Why is mAP the standard metric?

In EP01_02, you saw that the choice of threshold significantly alters Precision and Recall. The **mAP (*Mean Average Precision*)** addresses this: it evaluates the model at **multiple thresholds** (each threshold should generate a different confusion matrix) and summarizes performance by the **area under the Precision-Recall (P-R) curve**.

While the F1-Score examines a single equilibrium point, mAP considers the entire curve. The closer to **1.0**, the better the detector across all thresholds and classes (e.g., coins of 25, 50, and 1 real).

Metric	What it summarizes	Limitation
F1-Score	P × R balance at a single threshold	Depends on the chosen threshold
AP	Area under the P-R curve for one class	Valid only for a single class
mAP	Average of APs across all classes	More complex to implement

References: Roboflow — mAP · Explanatory video

1.18.6.2 How mAP is calculated — step by step

1. **Fixed thresholds** (always use this list):

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

2. For each threshold (t), classify the samples: `predito = 1 if confiança  t, else 0`. Compute TP, FP, FN, TN and obtain Precision((t)) and Recall((t)).
3. Build the P-R curve: pairs (Recall((t)), Precision((t))), ordered by increasing Recall.
4. **Monotonize Precision**:

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

5. Compute the **AP** (area under the monotonic curve) using the **trapezoidal rule** (a more accurate approximation than the simple Riemann sum):

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

6. **mAP** = average of the APs across all classes. In this assignment, there is only **1 class**, so mAP = AP.

> **Note**
>
> 📐 **Summary of the difference:**
> The *Riemann sum* approximates the area using rectangles, which may underestimate or overestimate. The **trapezoidal rule** uses trapezoids, reducing error by considering the average of the values at the interval endpoints, and is generally more accurate for piecewise smooth functions, such as the Precision-Recall curve.

1.18.6.3 📋 Task

Read an integer `n` (number of samples). Then read `n` lines, each containing: `true` (0 or 1) and `confidence` (float 0.0–1.0).

Calculate and print, for the **threshold 0.85** (index 9 in the list):

- Confusion Matrix (TP, FN, FP, TN)
- Accuracy, Precision, Recall, and F1-Score

Then, for **all thresholds**, print:

- Raw Precisions, monotonic Precisions, and Recalls, separated by `,`
- Final mAP

1.18.6.4 📌 Important

- Fixed threshold for the individual metrics: **0.85**
- Safe division: if the denominator is zero, use 0
- Formatting: two decimal places
- Monotonize from back to front
- Figure 1.13 presents a simulation of this problem

1.18.6.5 📌 Example Run

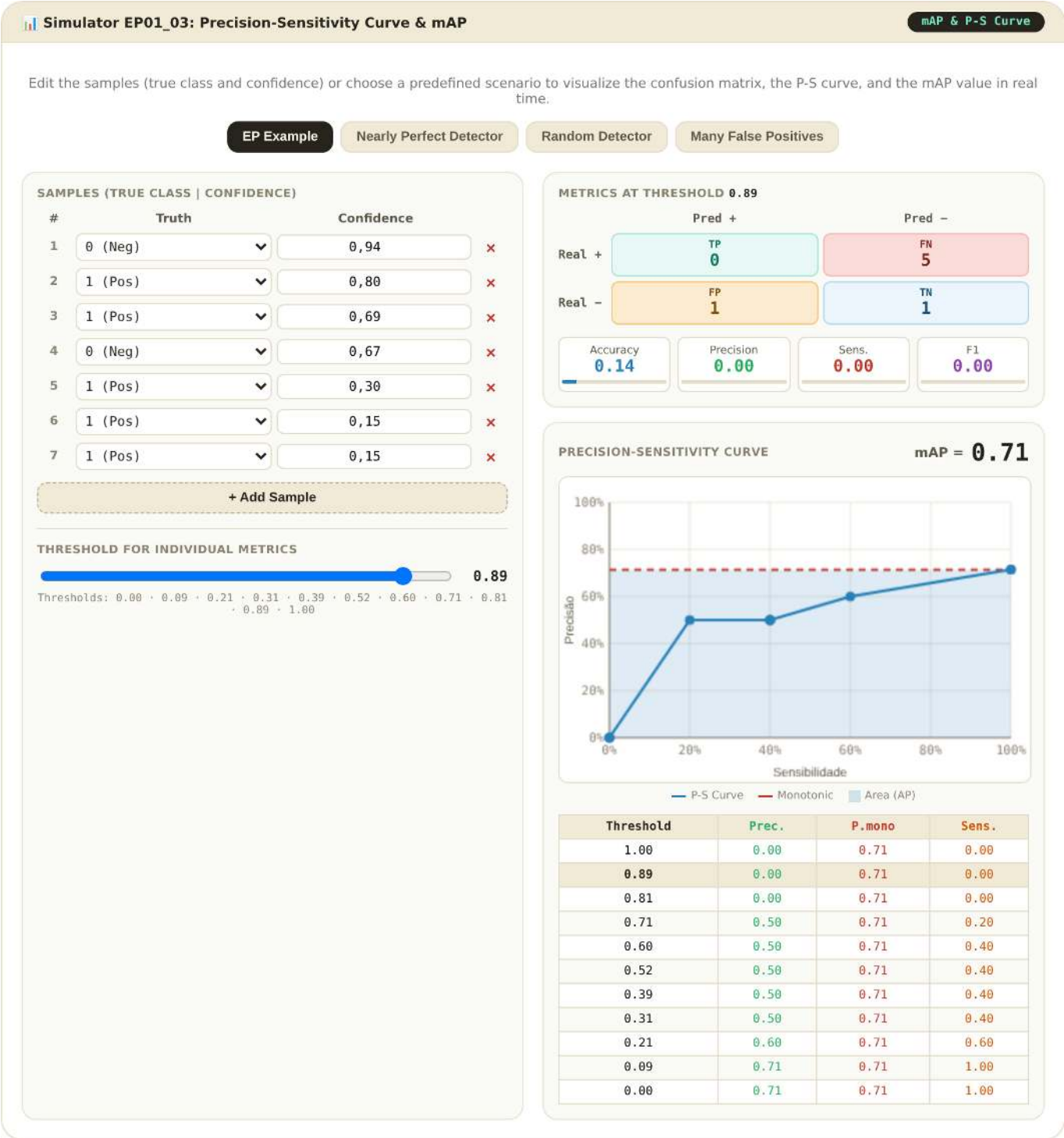
Input	Expected Output
7 0 0.94 1 0.80 1 0.69 0 0.67 1 0.30 1 0.15 1 0.15	# METRICS FOR THRESHOLD 0.85 # Confusion Matrix: TP = 0, FN = 5 FP = 1, TN = 1 Evaluation Metrics: Accuracy: 0.14 Precision: 0.00 Recall: 0.00 F1-Score: 0.00 # METRICS FOR ALL THRESHOLDS # Precisions: 0.00, 0.00, 0.00, 0.50, 0.50, 0.50, 0.50, 0.50, 0.60, 0.71, 0.71 Monotonic Precisions: 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71 Recalls: 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

1.18.6.6 🐍 Tip for calculating AP (with trapezoid rule)

```python
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):
        # Trapezoid rule: average of heights times the 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
```

```python
# your solution
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap01/sim-ep0103-map.html

Figure 1.13: Simulator EP01_03: Mean Average Precision (mAP) and P-R Curve

```
<IPython.core.display.HTML object>
```

1.18.7 EP01_04 🖼️ Reading and Information from a Matrix Image

In this activity, you must write a program that processes a digital image represented as a matrix of grayscale pixels.

- Read two integers **L** and **C**, representing the number of rows and columns.
- Read the **L * C** integer values that compose the image matrix (each value between 0 and 255).
- Calculate and print the following information:

1. The number of rows.
2. The number of columns.
3. The value of the largest pixel (**Maximum**).
4. The value of the smallest pixel (**Minimum**).
5. The arithmetic **Mean** of all pixels.

📌 **Important**:

- The output must follow exactly the labeled format (e.g., `Linhas: X`).
- The mean value must be formatted with **two decimal places**.
- See an interactive simulator for this question at **Figure 1.14** (interactive grid for visualizing intensities and real-time calculations).

1.18.7.1 🧠 Why Does This Matter? – The Image as Data

Every digital image is, at its core, a data structure. In 8-bit grayscale, each pixel is a scalar value. Extracting basic statistics is the first step toward:

Operation	Practical Utility
Maximum/Minimum	Identifying whether the image is "washed out" (low contrast) or saturated.
Mean	Calculating the overall brightness of the scene for exposure adjustments.
Normalization	Rescaling values to ranges such as $[0, 1]$ in neural networks.

1.18.7.2 📋 Task (VPL specification)

Input:

The first line contains the integer `L` (rows).

The second line contains the integer `C` (columns).

The following lines contain the elements of the matrix.

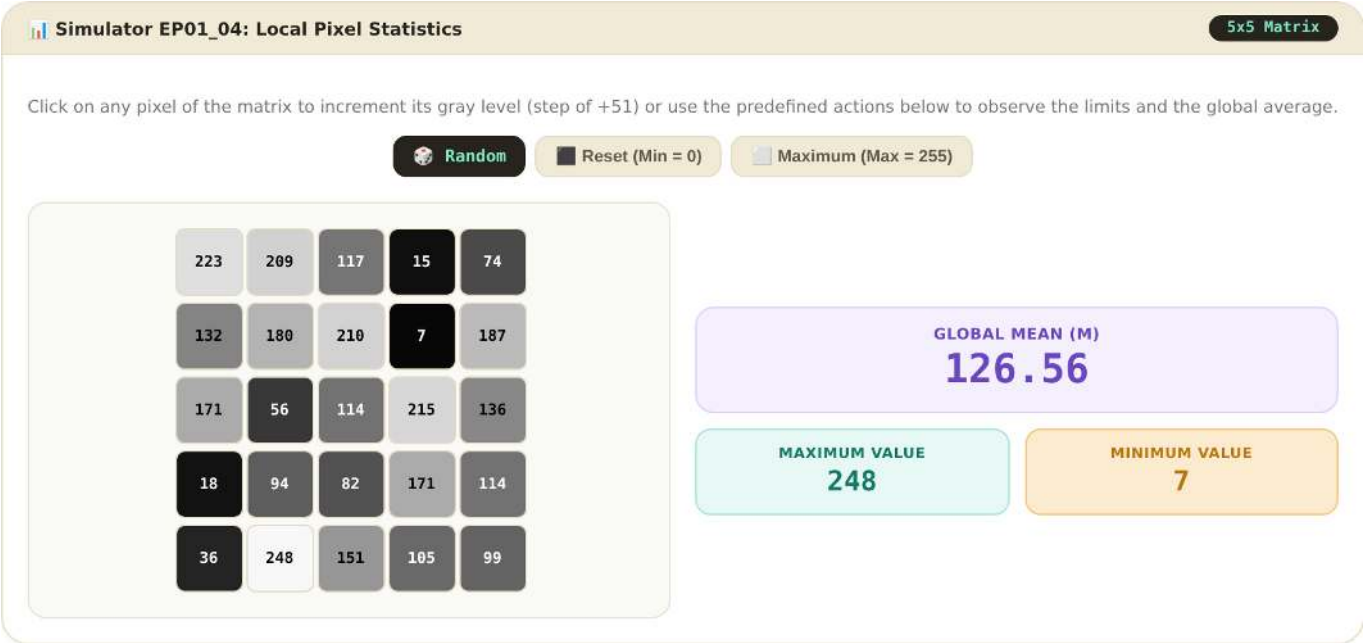
Output:

Five lines formatted according to the example:

```
Linhas: L

Colunas: C

Max: V

Min: V

Media: V.VV
```

1.18.7.3 📌 Examples

Input	Output	Observation
2 3 0 128 255 50 100 200	Linhas: 2 Colunas: 3 Max: 255 Min: 0 Media: 122.17	Small high-contrast image



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap01/sim-ep0104-estatisticas.html

Figure 1.14: EP01_04 Simulator: Pixel Statistics in Discrete 5x5 Matrix

```
# your solution
```

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

```
<IPython.core.display.HTML object>
```

1.18.8 EP01_05 🔄 Negative of a Grayscale Image

In this activity, you must write a program that computes the negative of a digital image.

- Read two integers **L** and **C**, representing the number of rows and columns.
- Read the integer values that make up the image matrix.
- For each pixel, apply the inversion transformation:

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

- Print the resulting matrix, preserving the original format (L rows and C columns).

📌 **Important**:

- The values in each row of the output must be separated by a **single space**.
- The output must contain **only the numbers** of the resulting matrix.

- See an interactive simulator for this problem at **Figure 1.15** (real-time comparison between the original matrix and its negative).

1.18.8.1 🧠 Why Does This Matter? – Intensity Inversion

The **negative** is a basic linear transformation that inverts the brightness scale. It is an essential tool for the human eye to identify light details that are "hidden" in darker backgrounds, being widely used in:

Application	Utility
Medical Imaging	Enhances the visualization of anomalies in dense tissues (e.g., X-rays).
Astronomy	Highlights faint galaxies and nebulae against the void of space.
Digital Arts	Aesthetic effects and preparation of selection masks.

1.18.8.2 📋 Task (Specification for VPL)

Input:

The first line contains the integer `L`.

The second line contains the integer `C`.

The following lines contain the elements of the matrix.

Output:

The inverted matrix with `L` rows and `C` columns.

1.18.8.3 📌 Examples

Input	Output	Observation
2 3 0 128 255 50 100 255	255 127 0 205 155 55	Where it was 0 (black) becomes 255 (white)

```
# your solution
```

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

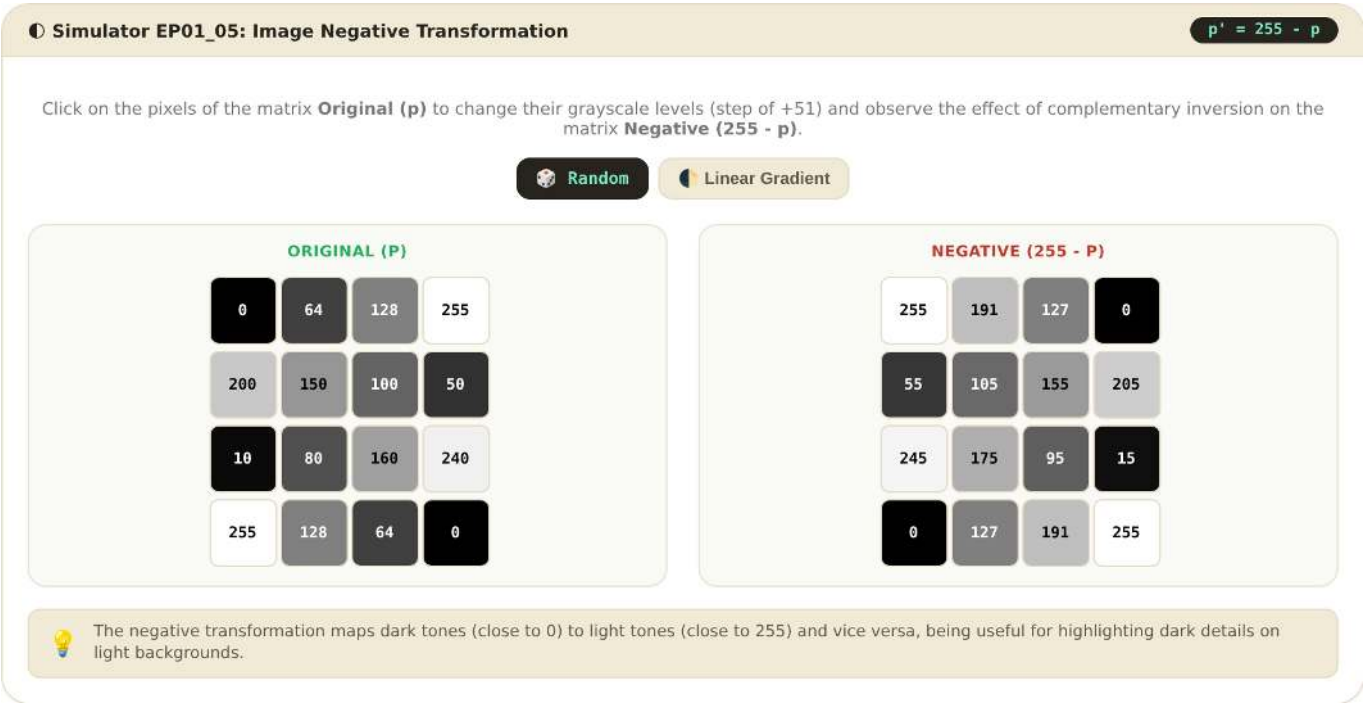
```
<IPython.core.display.HTML object>
```

1.18.9 EP01_06 🎨 RGB → Grayscale Conversion (ITU-R BT.601)

In this activity, you must write a program that converts colored pixels (RGB) to grayscale using the physiological weighting of the ITU-R BT.601 standard.

- Read two integers **L** and **C**, representing the number of rows and columns.
- Read **L** × **C** triples of integers, where each triple represents the **R** (Red), **G** (Green), and **B** (Blue) channels of a pixel.
- For each pixel, compute the grayscale value (g) using the formula:

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap01/sim-ep0105-negativo.html

Figure 1.15: EP01_05 Simulator: Negative Image Transformation (Intensity Inversion)

- Print the resulting matrix (L rows and C columns) containing the converted integer values.

📌 **Important**:

- Use the `round()` function from your language to ensure correct rounding to the nearest integer.
- The output should contain only the grayscale values, preserving the matrix structure (separated by spaces within each row).
- See an interactive simulator for this problem at **Figure 1.16** (adjust the sliders to see how each color contributes to the final brightness).

1.18.9.1 🧠 Why not just use the average?

The human eye does not perceive all colors with the same intensity. We are much more sensitive to **Green** than to **Blue** due to our biological evolution. The ITU-R BT.601 standard uses specific weights to create a grayscale image that appears naturally correct to our vision:

Channel	Weight	Human Perception
🟢 **Green**	**58.7%**	Maximum sensitivity (distinguishing foliage).
🔴 **Red**	**29.9%**	Medium sensitivity.
🔵 **Blue**	**11.4%**	Low sensitivity (darker shades).

1.18.9.2 📋 Task (VPL specification)

Input:

The first line contains the integer `L`.

The second line contains the integer `C`.

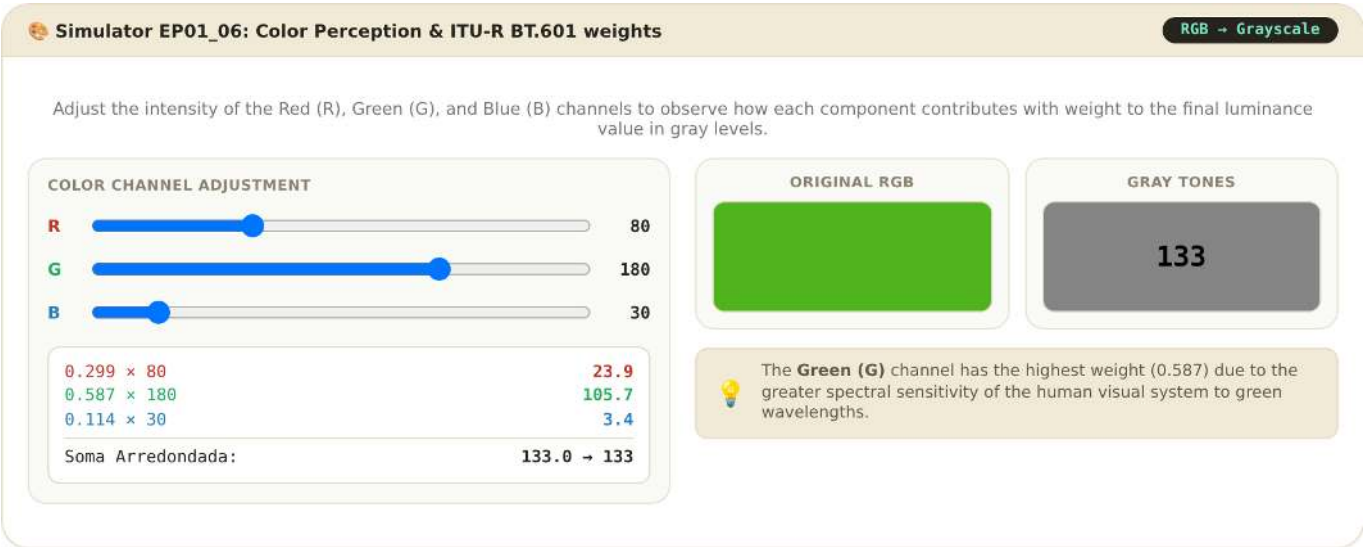
The following lines contain triples of integers `R G B` for each pixel.

Output:

The grayscale matrix with `L` rows and `C` columns.

1.18.9.3 📌 Examples

Input	Output	Observation
1 3 255 0 0 0 255 0 0 0 255	76 150 29	Note how Green (150) is brighter than Blue (29)



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap01/sim-ep0106-rgb-gray.html

Figure 1.16: EP01_06 Simulator: RGB to Grayscale Conversion (Perceptual ITU-R BT.601 Weighting)

```
# your solution
```

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

```
<IPython.core.display.HTML object>
```

1.18.10 EP01_07 ⚫ Manual Thresholding: Binary Image

In this activity, you must write a program that performs image segmentation through thresholding.

- Read two integers **L** and **C**, representing the dimensions of the matrix.
- Read an integer **T**, which will be the threshold (cutoff) value.
- Read the integer values of the matrix.
- For each pixel p, apply the following binarization rule:

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

- Print the resulting matrix containing only the values 0 or 255.

📌 **Important**:

- Pay attention to the operator: the pixel only becomes white (255) if it is **strictly greater** than T.
- The output must preserve the matrix structure (L rows and C columns).
- See an interactive simulator for this problem at **Figure 1.17** (adjust the T slider to observe how objects are isolated from the background).

1.18.10.1 🧠 What is Segmentation?

Thresholding is the simplest method for separating objects of interest from the image background. By converting grayscale tones into pure black and white, we create a binary map that facilitates object counting or shape identification:

Pixel Value (p)	Condition	Final Result
Dark ($p \leq T$)	Background/Noise	**0** (Black)
Light ($p > T$)	Object/Highlight	**255** (White)

1.18.10.2 📋 Task (specification for VPL)

Input:

The first line contains the integer `L`.

The second line contains the integer `C`.

The third line contains the integer `T` (threshold).

The following lines contain the elements of the matrix.

Output:

The binarized matrix (0 or 255) with `L` rows and `C` columns.

1.18.10.3 📌 Examples

Input	Output	Observation
2 4 128 0 100 128 200 50 129 255 64	0 0 0 255 0 255 255 0	Note that the value 128 became 0 (since $128 \leq 128$)

```
# your solution
```

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

```
<IPython.core.display.HTML object>
```

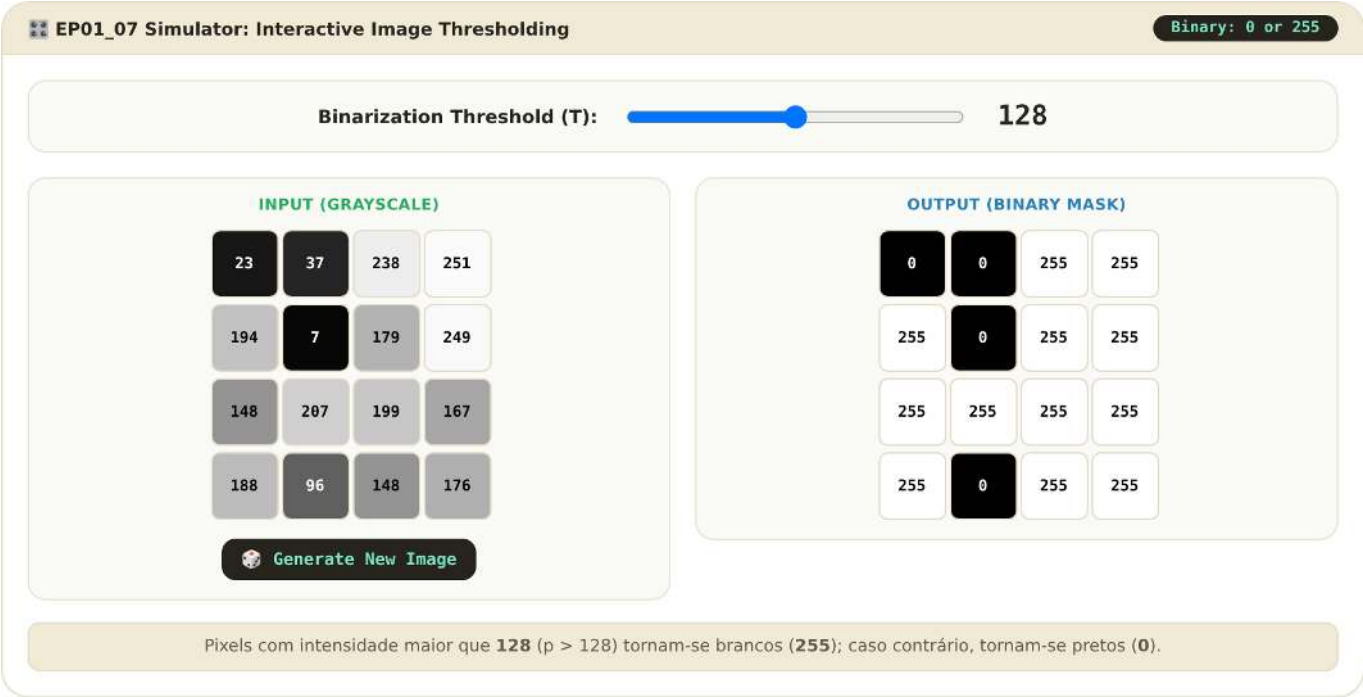
1.18.11 EP01_08 🎨 Intensity Range Remapping

In this activity, you must write a program that applies distinct linear transformations to different intensity regions of the image.

- Read two integers **L** and **C**, representing the matrix dimensions.
- Read the integers **T** (threshold), (delta 1), and (delta 2).
- Read the integer values of the matrix.
- For each pixel p, apply the conditional remapping rule:

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

- Print the resulting matrix with the new intensity values.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap01/sim-ep0107-limiarizacao.html

Figure 1.17: Simulator EP01_07: Interactive Global Thresholding (Binarization p > T)

📌 **Important**:

- is the offset applied to dark pixels (below the threshold).
- is the offset applied to bright pixels (greater than or equal to the threshold).
- Test cases guarantee that the result will always be within the valid range of **0 to 255**, so there is no need to handle saturation or rounding.
- The output must preserve the matrix structure (**L** rows and **C** columns).

1.18.11.1 🧠 Conditional Pixel Transformation

In Digital Image Processing (DIP), we often need to handle regions independently. This technique allows, for example, brightening only the shadows of a photograph (increasing dark pixels) without blowing out the highlights of already bright areas, or vice versa.

Intensity Range	Condition	Operation
Dark Pixels	$p < T$	$p + \delta_1$
Bright Pixels	$p \geq T$	$p + \delta_2$

1.18.11.2 📋 Task (VPL specification)

Input:

The first line contains the integers `L` and `C`.

The second line contains the integers `T`, , and .

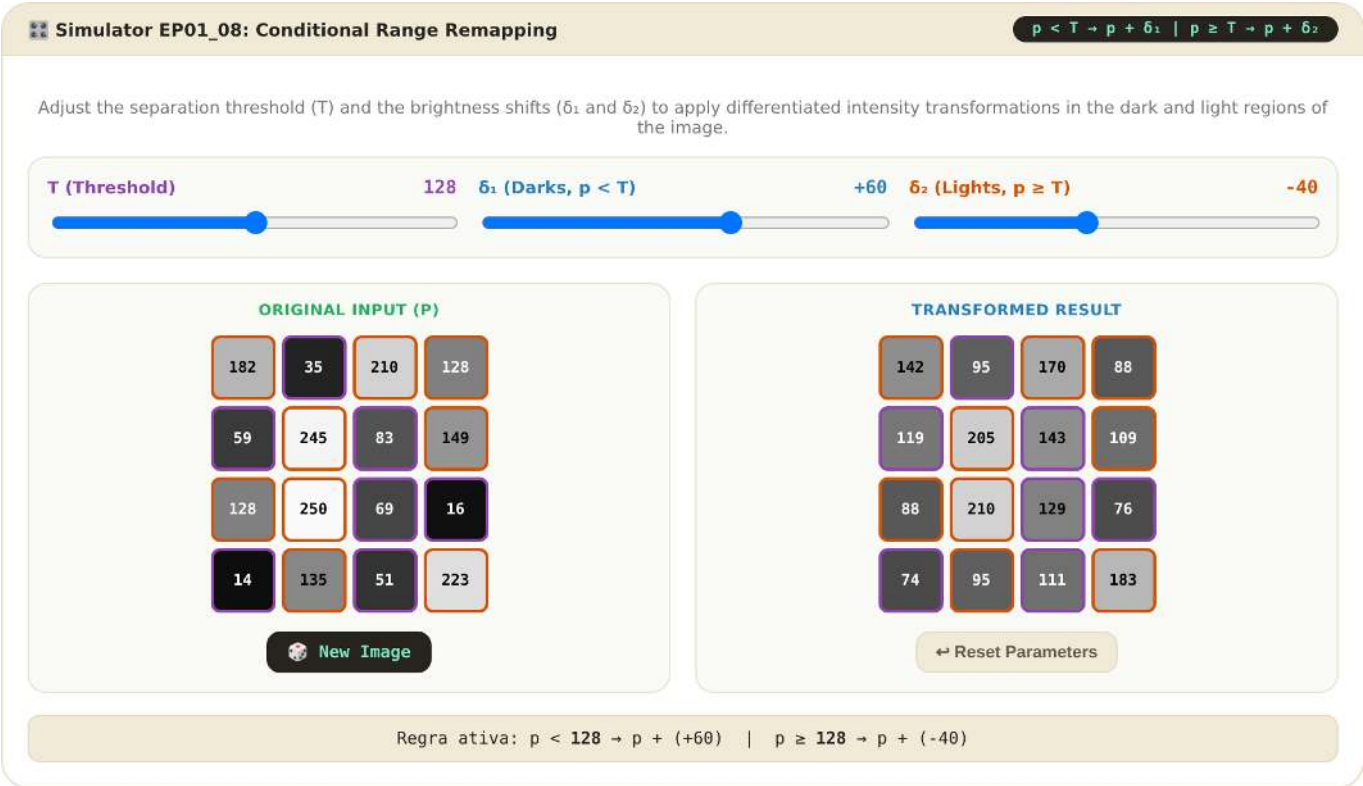
The following lines contain the matrix elements.

Output:

The transformed matrix with `L` rows and `C` columns, with space-separated values.

1.18.11.3 📌 Examples

Input	Output	Observation
2 4 128 60 -40 0 100 150 255 80 128 200 30	60 160 110 215 140 88 160 90	Pixels < 128 add 60. Pixels ≥ 128 subtract 40.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap01/sim-ep0108-brilho-contraste.html

Figure 1.18: EP01_08 Simulator: Conditional Range Remapping (Brightness and Contrast by Threshold)

```
# your solution
```

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

```
<IPython.core.display.HTML object>
```

1.18.12 EP01_09 🏁 Checkerboard Pattern: Chess Matrix

In this activity, you must write a program that generates a synthetic image in the pattern of a chessboard.

- Read two integers **L** (rows) and **C** (columns).
- Generate a matrix where the values alternate between **0** (black) and **1** (white).
- The filling logic must follow the parity rule:
- The element at position $(0, 0)$ is always **0**.
- A pixel at position (i, j) will be **1** if the sum of the indices $(i + j)$ is **odd**.
- A pixel at position (i, j) will be **0** if the sum of the indices $(i + j)$ is **even**.

📌 **Important**:

- The colors must alternate correctly both horizontally and vertically.

- The output must be the matrix printed line by line, with the elements separated by a space.
- See an interactive simulator for this problem at **Figure 1.19** (adjust the dimensions to visualize the construction of the grid and the corresponding textual output).

1.18.12.1 🧠 Synthetic Patterns

Creating geometric patterns is a fundamental exercise for mastering **index logic** in matrices. In Digital Image Processing, the checkerboard pattern is not merely aesthetic; it is widely used for:

Application	Utility
Camera Calibration	Estimating intrinsic and extrinsic lens parameters.
Distortion Correction	Identifying and correcting the "barrel" or "pincushion" effect in wide-angle lenses.
3D Mapping	Projecting known patterns to reconstruct surfaces in structured light systems.

1.18.12.2 📋 Task (specification for VPL)

Input:

A line containing the integer `L` (rows).

A line containing the integer `C` (columns).

Output:

The checkerboard matrix with `L` rows and `C` columns, printed with spaces between the elements.

1.18.12.3 📌 Examples

Input	Output	Observation
3 4	0 1 0 1 1 0 1 0 0 1 0 1	Note that each row starts with the inverse of the previous one

```
# your solution
```

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

```
<IPython.core.display.HTML object>
```

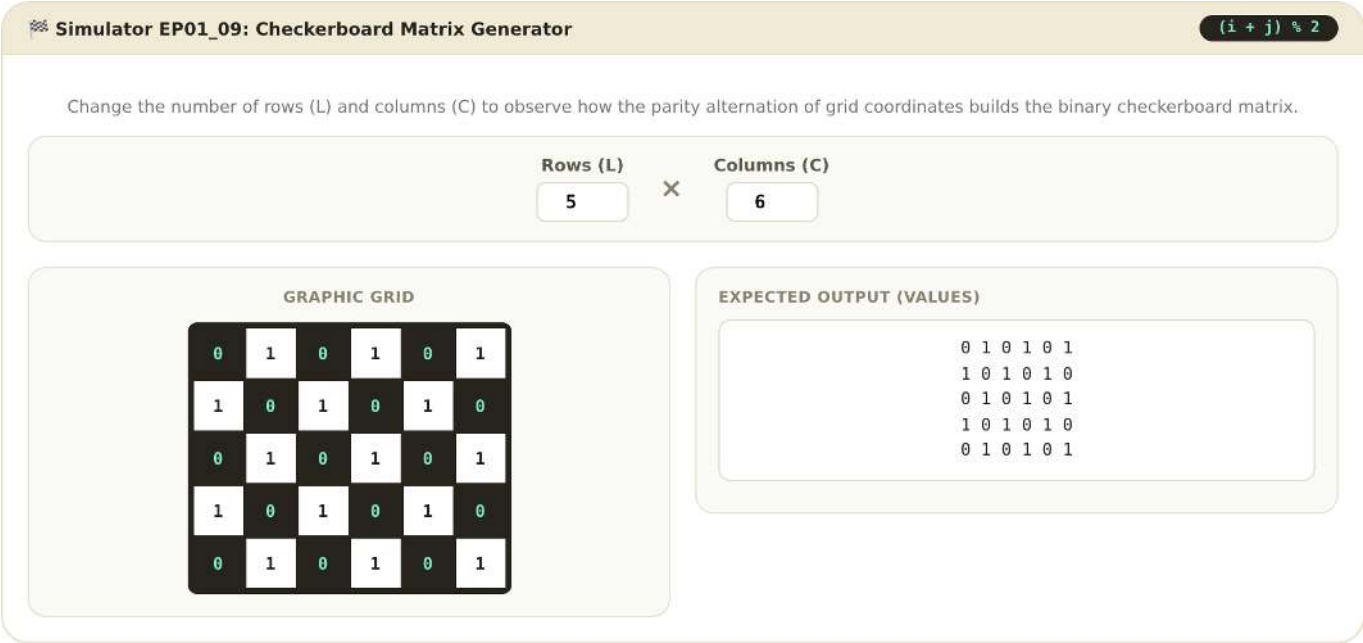
1.18.13 EP01_10 📄 Metadata: Reading a PGM File

In this activity, you must read an image file in **PGM (Portable Gray Map)** format and extract its dimensions from the header.

- The **PGM (P2)** format is a plain text (ASCII) file that stores grayscale images.
- The file has a **header** structured as follows:

1. **Version:** The identifier `P2`.
2. **Comments:** Optional lines starting with `#` (should be ignored).
3. **Dimensions:** Two integers representing **Width** and **Height**.
4. **Maximum:** An integer representing the maximum intensity (usually 255).

- After the header, the pixel data follows.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap01/sim-ep0109-xadrez.html

Figure 1.19: EP01_09 Simulator: Checkerboard Pattern Generator (Parity Logic (i + j) % 2)

📌 **Important**:

- **File Reading:** You must open the file indicated in the example using Python's `open()` function.
- **Output Order:** Contrary to the order present in the file, the expected output must be in tuple format: `(Height, Width, Channels)`.
- Since PGM files are grayscale, the number of **Channels** is always **1**.
- See an interactive simulator for this question at **Figure 1.20** (adjust the dimensions to see how the ASCII header is generated).

1.18.13.1 🧠 Understanding the PGM Format

The PGM format is one of the simplest for image processing. Because it is plain text, it allows you to view the metadata and even the pixel values by opening the file in a notepad:

Component	Example	Meaning
Magic Number	`P2`	Identifies that it is a PGM in text format (ASCII).
Comment	`# CREATOR...`	Informational line ignored by the processor.
Dimensions	`397 343`	**397** columns (Width) and **343** rows (Height).
Intensity	`255`	Defines the value of pure white (scale from 0 to 255).

1.18.13.2 📋 Task (VPL specification)

Input:

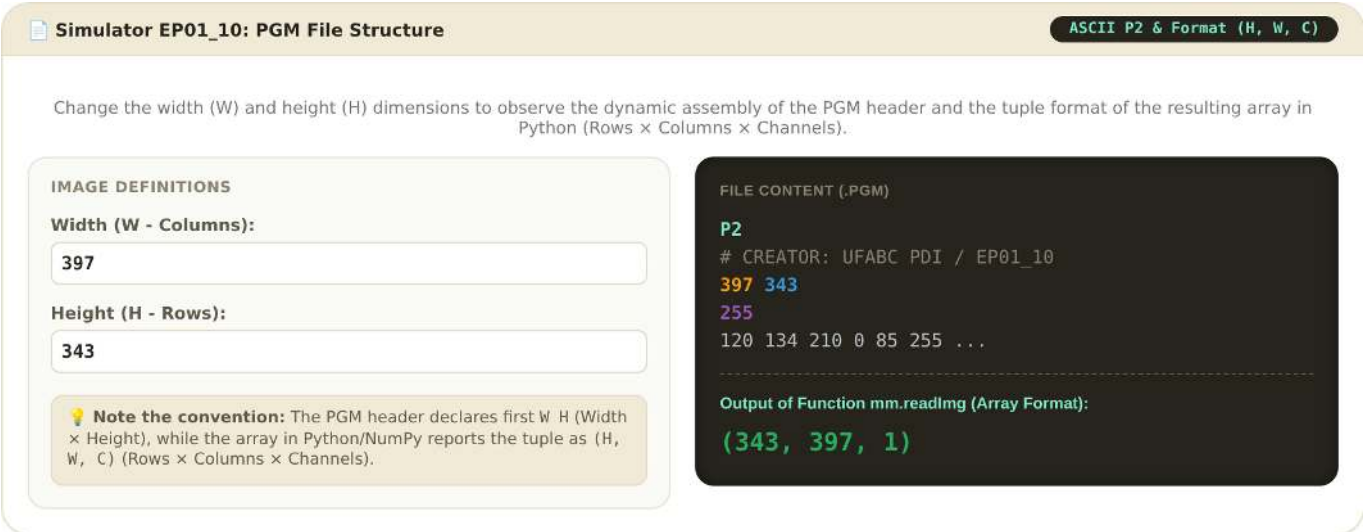
No keyboard input. The program must read the file `"aula01fig03b.pgm"` present in the execution directory.

Output:

A tuple containing `(Height, Width, 1)`.

1.18.13.3 📌 Examples

File Name	Expected Output	Note
“aula01fig03b.pgm”	(343, 397, 1)	Note the order inversion: Height first



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap01/sim-ep0110-pgm.html

Figure 1.20: Simulator EP01_10: PGM File Structure (ASCII P2 Header and Mapping to Python Tuple)

📝 **Note**

The file required for this EP will be automatically downloaded from the repository using the following code:

```
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"  File downloaded: {f}")
        except urllib.error.HTTPError as e:
            print(f"  Error {e.code}: Not found at {url}")
```

```
# your solution
```

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

```
<IPython.core.display.HTML object>
```

1.18.14 EP01_11 ↗ Neighborhood Analysis: 1D Maximum Filter

In this activity, you must implement a simple morphological maximum filter operating on a one-dimensional signal (vector).

- Read an integer **n**, representing the size of the vector.
- Read the **n** integer elements that make up the original vector **v1**.
- Create a new vector **v2**, where each position i is the result of comparing the current element with its immediate neighbors:

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

📌 **Important**:

- **Boundaries:** At the ends of the vector (indices 0 and $n-1$), the neighborhood has only two elements (the element itself and the only available neighbor). At index 0, compare only $v1[0]$ and $v1[1]$. At the last index, compare only $v1[n-2]$ and $v1[n-1]$.
- **Output:** Print the header "v2:" followed by the values of the resulting vector, one per line.
- See an interactive simulator for this question at **Figure 1.21** (hover over the results to view the neighborhood window used in the calculation).

1.18.14.1 🧠 Why analyze neighbors?

In image processing, the value of a pixel is rarely isolated; it depends on the context around it. The **Maximum Filter** is the basis of the **Dilation** operation in mathematical morphology, serving to:

Function	Visual Effect
Enhancement	Expands bright structures and "thickens" light objects.
Noise Removal	Eliminates small black spots (dark "salt and pepper" noise).
Filling	Closes small holes or gaps in binary shapes.

1.18.14.2 📋 Task (VPL specification)

Input:

An integer `n`.

On the following lines, the `n` integer elements of the vector.

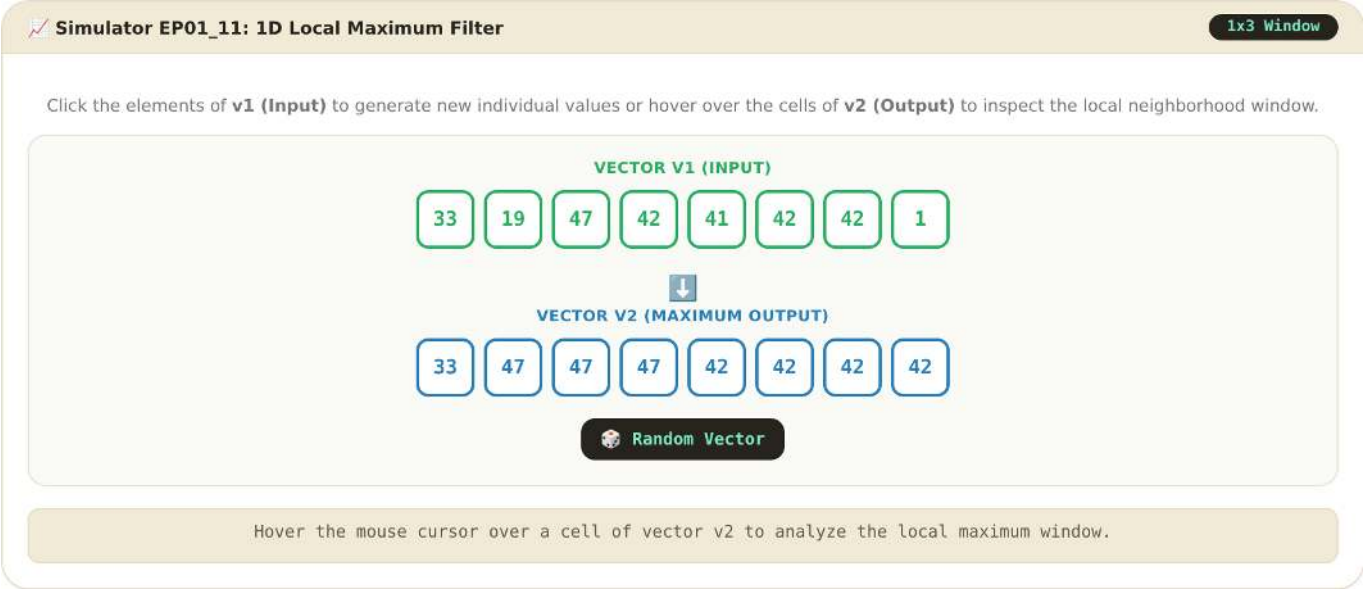
Output:

The string `v2:` on the first line.

On the following lines, each element of `v2` (one per line).

1.18.14.3 📌 Examples

Input	Output	Observation
5	v2:	At index 1: max(10, 20, 5) = 20
10	20	
20	20	
5	30	
30	30	
15	30	



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap01/sim-ep0111-filtro-maximo.html

Figure 1.21: EP01_11 Simulator: 1D Local Maximum Filter (1x3 Neighborhood with Border Condition)

```
# your solution
```

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

```
<IPython.core.display.HTML object>
```

Chapter 2

From Capture to Pixel - Sampling, Quantization, and Connectivity

cap02/images/thumbnail.jpg cap02/images/abstract.jpg

This chapter deepens the understanding of the digital image, transitioning from the physical nature of capture to its discrete mathematical representation. We investigate how light becomes data and how the spatial organization of pixels defines the neighborhood relations and connectivity essential for advanced Computer Vision (CV) algorithms.

2.1 Objectives

At the end of this chapter, you will be able to:

- Explain the physical model of image formation based on illumination and reflectance.
- Differentiate the mechanisms of human vision and digital sensors.
- Understand the processes of **sampling** (discretization of space) and **quantization** (discretization of intensity).
- Describe the topological relationships between pixels: neighborhood, adjacency, connectivity, and distances.
- Perform basic geometric transformations (translation, rotation, scaling) while preserving quality.
- Visualize in practice the effects of varying spatial resolution and bit depth.

2.2 The Eye and the Camera - Elements of Visual Perception

The formation of a digital image begins with the capture of light reflected by objects. As illustrated in Figure 2.1, both the human eye and digital cameras follow similar optical principles to focus light onto a sensitive surface, although they employ distinct biological and electronic mechanisms for signal transduction.

2.2.1 Human vision

The eye functions as a complex optical system: light passes through the cornea, the aqueous humor, the pupil (controlled by the iris), and the lens — which adjusts focus dynamically — until it reaches the retina. In the retina, photoreceptors are found: the **cones** (6 million), concentrated in the fovea, are responsible for color and detail vision, while the **rods** (120 million) ensure vision under low illumination (scotopic vision), detecting only shades of gray. The blind spot is the region from which the optic nerve originates, lacking receptors.

2.2.2 Digital sensors

In cameras, image sensors play the role of the retina. The two most common types are the **CCD** (*Charge-Coupled Device*) and the **CMOS** (*Complementary Metal-Oxide-Semiconductor*). The sensor is composed of an array of photosites (pixels) that accumulate electrical charge proportional to the incident light.

For color reconstruction, the **Bayer Filter** is used, a matrix of color filters that allows each pixel to capture only one color component: red, green, or blue (RGGB - *Red, Green, Green, Blue*). Subsequently, an **ADC** (*Analog-to-Digital Converter*) quantizes this charge into numerical values, defined by a bit depth (e.g., 8 bits, resulting in 256 intensity levels).

> **Curiosity**
>
> Although the human eye has millions of receptors, high-definition resolution is confined to the fovea (central vision), equivalent to approximately 120×120 pixels. The perception of a complete scene in high resolution is the result of intensive post-processing performed by the brain.

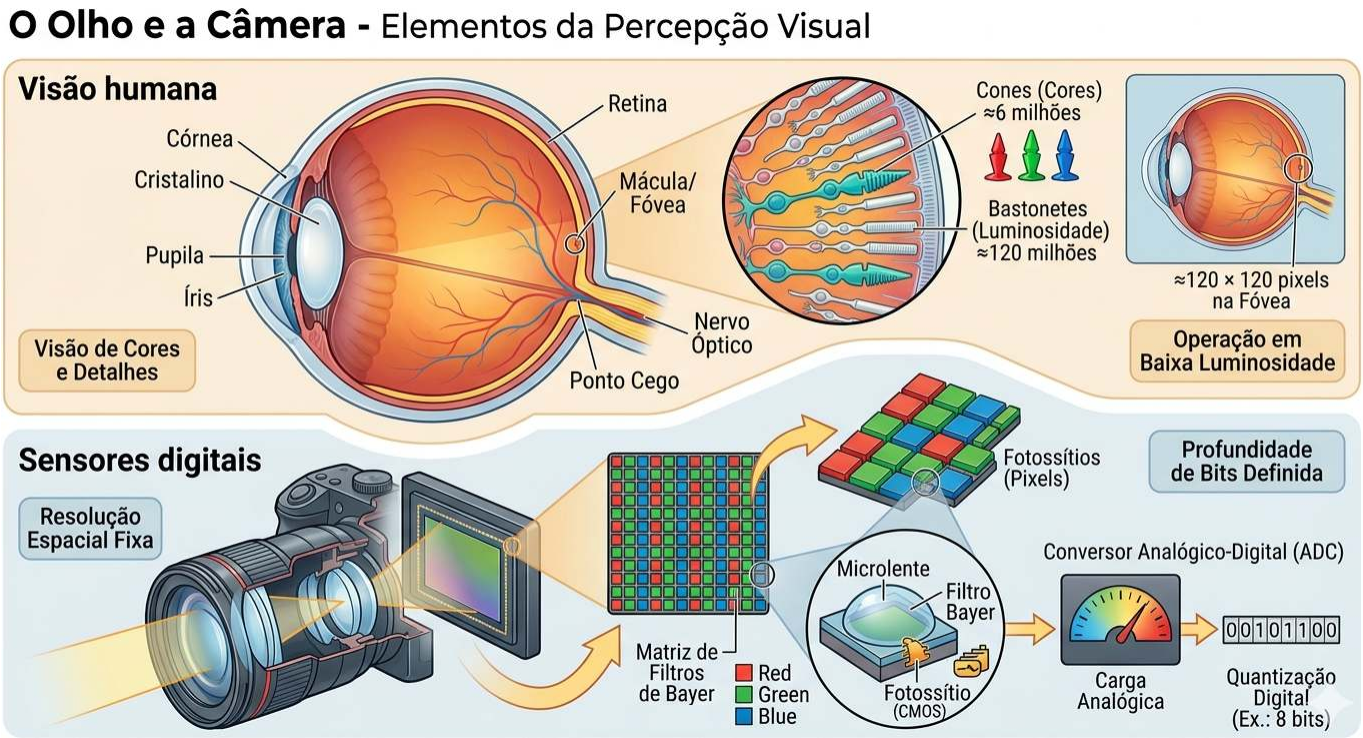


Figure 2.1: Didactic comparison between the biological and electronic visual systems: at the top, the anatomy of the human eye highlighting the retina and photoreceptors (cones and rods); below, the structure of a digital camera detailing the CMOS sensor, the Bayer filter array (RGGB), and the digital quantization process carried out by the ADC.

2.3 Optical Illusions: The Challenges of Visual Perception

While digital sensors capture light intensity in a linear and objective manner, the human visual system interprets the scene based on context, prior experiences, and biological survival mechanisms. Optical illusions are not "errors" of the eye but evidence of the intense **brain post-processing** carried out in the visual cortex.

2.3.1 Ambiguity and Context

The brain constantly seeks to make sense of ambiguous patterns. In the example of the **Rubin Vase** (see Figure 2.2), perception alternates between the figure (vase) and the background (two faces), demonstrating that we cannot process both interpretations simultaneously. The **Shepard Elephant** illusion, in turn, plays on our inability to reconcile contour lines that suggest volume in logically impossible positions.

2.3.2 Brightness and Local Contrast

Many illusions arise from **lateral inhibition**, a mechanism by which neighboring neurons in the retina compete with one another to enhance edges. In the **Scintillating Grid**, "ghostly" dark spots seem to appear at the white intersections due to this local contrast processing.

The **Adelson Checker Shadow** illusion is perhaps the most striking for computer vision: square "A" and square "B" have exactly the same gray value in the sensor (or digital file), but the brain "corrects" the brightness of "B" by understanding that it lies under a cast shadow, perceiving it as lighter.

2.3.3 Geometry and Perspective

Depth perception can be deceived by geometric constructions that challenge three-dimensional logic from a specific viewing angle. The **Schröder Staircase** exploits perspective ambiguity to create an object that appears to ascend or descend depending on how it is observed, highlighting how our interpretation of "up" and "down" depends on the vanishing point.

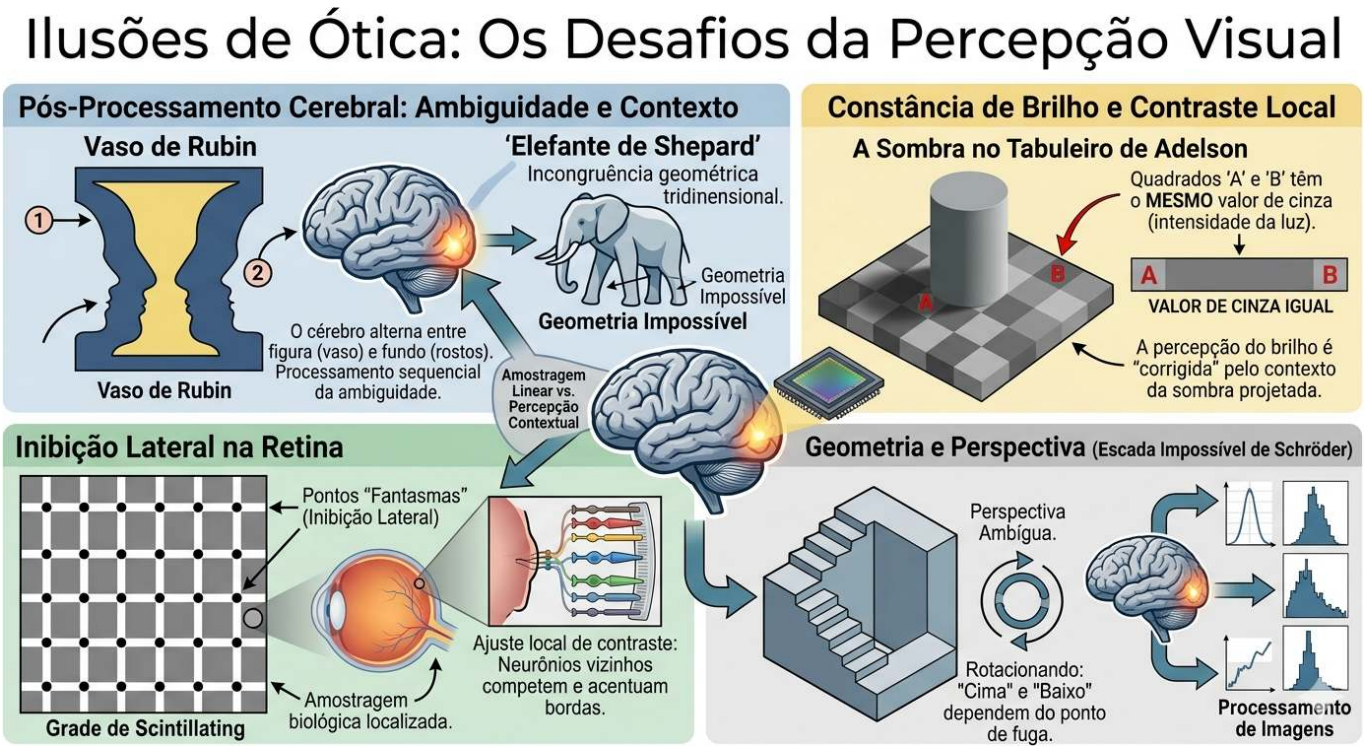


Figure 2.2: Collection of perceptual challenges: (top left) Rubin Vase — figure-ground ambiguity; (top center) Shepard Elephant — geometric incongruity; (top right) Adelson Shadow — brightness constancy based on context; (bottom left) Dot grid — lateral inhibition; (bottom right) Schröder Staircase.

2.4 The Mathematical Model of Image Formation

An image can be modeled as the product of two functions:

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

where:

- $i(x, y)$ is the **illumination** incident upon the scene (light energy per unit area), determined by the light source;
- $r(x, y)$ is the **reflectance** of the object (fraction of reflected light), determined by the optical properties of the surface, with $0 < r(x, y) < 1$.

In practice, the two components vary at distinct spatial scales: $i(x, y)$ tends to vary slowly across space, whereas $r(x, y)$ may exhibit abrupt variations associated with textures, edges, and fine details. Digital image processing (DIP) techniques often seek to separate or compensate for these components, as in methods for non-uniform illumination correction.

Figure 2.3 illustrates how this model manifests in digital color images, represented by multiple spectral channels (RGB) and, optionally, by an additional transparency channel (RGBA).

2.4.1 Digital Representation and Spectral Channels

In digital color images, the function $f(x, y)$ from Equation 2.1 is represented by multiple spectral channels. In the **RGB** standard, each pixel stores three independent samples:

$$\mathbf{f}(x, y) = [R(x, y),\ G(x, y),\ B(x, y)]$$

corresponding to the intensities of the red, green, and blue components. In **RGBA** images, a fourth channel is added:

$$\mathbf{f}(x, y) = [R(x, y),\ G(x, y),\ B(x, y),\ A(x, y)]$$

where $A(x, y)$ represents the **alpha** channel, responsible for encoding the pixel's opacity. By convention, the value $A = 0$ indicates a fully transparent pixel, while $A = 255$ (or 1) represents a fully opaque pixel.

Thus, a digital color image is structured as a multidimensional matrix with dimensions:

- **RGB:** $M \times N \times 3$
- **RGBA:** $M \times N \times 4$

where each position (x, y) stores the samples associated with the optical properties of that spatial coordinate.

The conversion between intensity ranges is straightforward and given by:

$$f_{\text{norm}}(x, y) = \frac{f(x, y)}{255}$$

where $f(x, y) \in [0,\ 255]$ and $f_{\text{norm}}(x, y) \in [0,\ 1]$.

Scale Convention Across Libraries: Different programming ecosystems adopt distinct value ranges and orderings to represent digital channels, as summarized in Table 2.1.

Table 2.1: Scale and channel order conventions in major DIP libraries.

Library	Default type	Value range	Band order	Array/tensor shape
OpenCV	`uint8`	$[0, 255]$	**BGR**	$H \times W \times C$
Pillow	`uint8`	$[0, 255]$	RGB	$H \times W \times C$
NumPy	`uint8/float32`	$[0, 255]$ or $[0, 1]$	—	$H \times W \times C$
scikit-image	`float64`	[0,0, 1,0]	RGB	$H \times W \times C$
PyTorch	`float32`	[0,0, 1,0]	RGB	$C \times H \times W$
TensorFlow/Keras	`float32`	[0,0, 1,0]	RGB	$H \times W \times C$

Neglecting these discrepancies is a frequent source of errors in CV workflows — such as, for example, feeding an image read via OpenCV (BGR default, `uint8` type) directly into a deep PyTorch model that assumes the normalized RGB format.

2.4.2 Preparing the Practical Environment

The following block loads the `morph.py` module from the repository and demonstrates, for a synthetic pixel, the different scaling conventions and channel orders adopted by the main image processing libraries.

```
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup(demo=True)
from morph import mm
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0

 Scale conventions by library
 NumPy / Pillow / OpenCV (uint8): [200 100  50] → [0, 255]
 scikit-image / PyTorch (float):  [0.78431373 0.39215686 0.19607843] → [0.0, 1.0]
 OpenCV: watch out - reads as BGR: [ 50 100 200] (channels reversed)
```

2.4.3 Image Reading Implementation in `morph.py`

The following excerpt displays the source code of the `mm.read` function, allowing you to directly verify how the `morph.py` library implements image reading.

```
import inspect
print(inspect.getsource(mm.read))
```

```
    @staticmethod
    def read(file, pil=False, grayscale=False):
        """Lê imagem (local, URL ou Google Drive) → PIL.Image, ndarray 2D ou RGB 3D."""
        import re, requests
        from PIL import Image
        from io import BytesIO
        from urllib.request import urlopen, Request
        np = mm._get_np()

        # - fonte: URL / Google Drive -
        if isinstance(file, str) and file.startswith(("http://", "https://", "id=")):
            m = re.search(r"id=([\w-]+)", file) or re.search(r"/d/([\w-]+)", file)
            url = f"https://drive.google.com/uc?export=view&id={m.group(1)}" \
                if m and ("id=" in file or "drive.google.com" in file) else file
            hdr = {"User-Agent": "Mozilla/5.0 AppleWebKit/537.36 Chrome/124 Safari/537.36"}
            try:
                r = requests.get(url, headers=hdr, timeout=20)
                if r.status_code == 429: raise requests.exceptions.HTTPError()
                r.raise_for_status()
                file = BytesIO(r.content)
            except:
                file = BytesIO(urlopen(Request(url, headers=hdr), timeout=20).read())

        img = Image.open(file)
        img.load()
        if pil:                     return img
        if grayscale:               return np.array(img.convert("L"))   # (H,W)
        if img.mode == "L":         return np.array(img)                # (H,W)
        if img.mode == "RGBA":                                          # fundo branco
            bg = Image.new("RGB", img.size, (255, 255, 255))
```

```
            bg.paste(img, mask=img.split()[3])
            return np.array(bg)
        return np.array(img.convert("RGB"))                          # (H,W,3)
```

2.4.4 RGB and RGBA: Practical Example with the Mandrill Image

The following example loads the Mandrill image in RGB format, artificially constructs an alpha channel with a horizontal gradient, and displays both representations, concretely illustrating the matrix structures $M \times N \times 3$ and $M \times N \times 4$.

```python
import os
import numpy as np

# https://commons.wikimedia.org/wiki/File:Mandrill-k-means.png
url    = "https://upload.wikimedia.org/wikipedia/commons/a/ab/Mandrill-k-means.png"
caminho = "imagens/mandrill.png"

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

img_rgb = np.array(img_pil.convert("RGB"))          # shape: (M, N, 3), uint8

#   Creation of the alpha channel (horizontal gradient)
h, w, _ = img_rgb.shape
alpha    = np.tile(np.linspace(0, 255, w, dtype=np.uint8), (h, 1))
img_rgba = np.dstack([img_rgb, alpha])               # shape: (M, N, 4), uint8

#   Diagnostics
print(f"RGB   → shape={img_rgb.shape}, dtype={img_rgb.dtype}, "
      f"range=[{img_rgb.min()}, {img_rgb.max()}]")
print(f"RGBA  → shape={img_rgba.shape}, dtype={img_rgba.dtype}, "
      f"range=[{img_rgba.min()}, {img_rgba.max()}]")
print(f"Pixel (0,0): RGB={img_rgb[0,0]}  |  RGBA={img_rgba[0,0]}")

#   Float32 normalization (scikit-image / PyTorch standard)
img_rgb_norm = img_rgb.astype(np.float32) / 255.0
print(f"\nNormalized (float32): range=[{img_rgb_norm.min():.2f}, {img_rgb_norm.max():.2f}]")
print(f"Pixel (0,0) normalized: {img_rgb_norm[0,0]}")

#   Visualization
mm.show(
    [img_rgb, img_rgba],
    title=["RGB  -  $M \\times N \\times 3$", "RGBA  -  $M \\times N \\times 4$"],
    cols=2, axis=True
)
```

```
RGB   → shape=(512, 512, 3), dtype=uint8, range=[14, 248]
RGBA  → shape=(512, 512, 4), dtype=uint8, range=[0, 255]
Pixel (0,0): RGB=[125 110  58]  |  RGBA=[125 110  58   0]

Normalized (float32): range=[0.05, 0.97]
Pixel (0,0) normalized: [0.49019608 0.43137255 0.22745098]
```

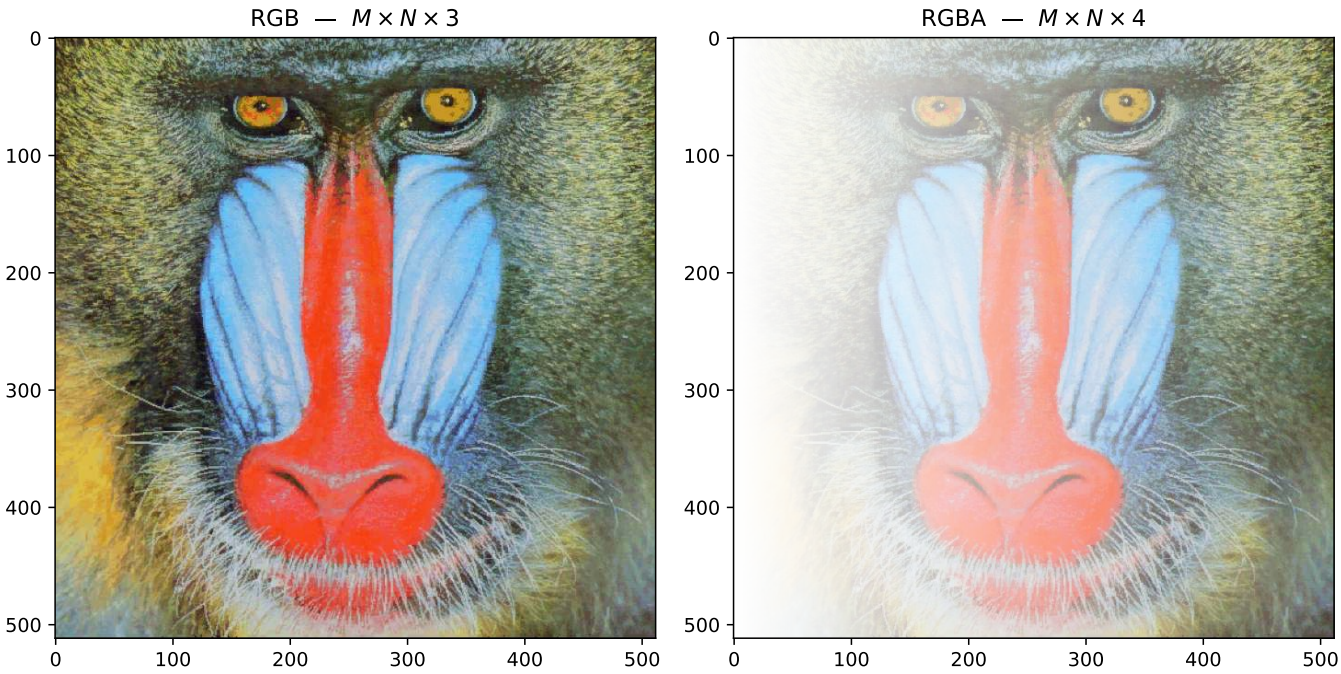


Figure 2.3: RGB image (3 channels, $M \times N \times 3$) and RGBA version (4 channels, $M \times N \times 4$) with gradual alpha transparency from left to right. Mandrill image — USC SIPI Image Database (public domain).

2.5 Digitization: Sampling and Quantization

To transform a continuous scene into a digital image, two processes are required: sampling and quantization.

2.5.1 Sampling - Discretization of Space

Sampling consists of measuring the value of the function $f(x, y)$ at equally spaced points, forming a matrix of M rows (height) and N columns (width). Each element of this matrix is a **pixel**. The **spatial resolution** is given by $M \times N$. The higher the resolution, the more spatial details are preserved, but also the greater the computational and storage cost.

2.5.2 Quantization - Discretization of Intensity

Quantization assigns to each pixel a discrete numerical value, usually represented by an integer of b bits. The **bit depth** defines the number of intensity levels: 2^b. Grayscale images typically use 8 bits (256 levels). Color images use three 8-bit channels (24 bits in total).

Illustration: If we use only 1 bit per pixel (black and white), we lose all intermediate tones. With 2 bits (4 levels), coarse gradients are already noticeable. With 8 bits, the human eye can hardly perceive the discretization (continuous vision).

> ⚠ Quantization Error
>
> **Quantization error** is the difference between the actual analog value and the assigned discrete value. It manifests as quantization noise, visible in regions with smooth gradients when few bits are used.

2.5.3 Effects of Sampling and Quantization

The following experiments show how reducing spatial resolution (subsampling) and bit depth degrade visual quality. Use the code to explore different factors and gray levels.

```
# Example image (striped-bearded) - basis for the sampling figures,
# quantization and geometric transformations of this chapter.
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)
    mm.write(mm.read(url), caminho)

img_color = mm.read(caminho)
img_gray0 = mm.gray(img_color)
img_gray  = mm.crop(img_gray0, 820, 1850, 890, 1550)  # crop to see details
print(f"Original image: {img_color.shape}")
```

```
Original image: (3067, 2047, 3)
```

```
import numpy as np

# Reinterpretation as a PIL object to access EXIF metadata (Python only).
img_pil   = mm.read(caminho, pil=True)   # returns PIL Image with EXIF
img_numpy = np.array(img_pil)
exif      = img_pil._getexif()           # now it works - img_pil is PIL Image
print(f"Image type: {type(img_color)}")
```

```
Image type: <class 'numpy.ndarray'>
```

The experiment presented in Figure 2.4 illustrates the trade-off between **spatial resolution** and **storage cost** in memory. The code employs the subsampling technique via *slicing* to reduce the original pixel matrix according to a factor f, resulting in memory savings — for example, a factor $f = 8$ reduces the data size by 64 times (8^2). For visual comparison purposes, the reduced images are restored to their original dimensions (512×512) through **nearest-neighbor** interpolation (`nearest`). This process does not recover the lost information, but it makes evident the effect of ***aliasing*** and the block structure (pixelation) generated by the low data density of the sampled matrix.

```
def subsample_simple(image, f):
    # Subsampling via slicing
    reduced = mm.subsample(image, f)

    # Memory calculation in KB
    mem_kb = reduced.nbytes / 1024
    label = f"{reduced.shape[1]}x{reduced.shape[0]}, {int(mem_kb)} KB\n(Fator {f})"

    # Restores the size for display (original H, W)
    res = mm.resize(reduced, (image.shape[1], image.shape[0]), method='nearest')
    return res, label

factors = [1, 4, 8, 12]

# Generates the results and splits them into lists for 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))
```

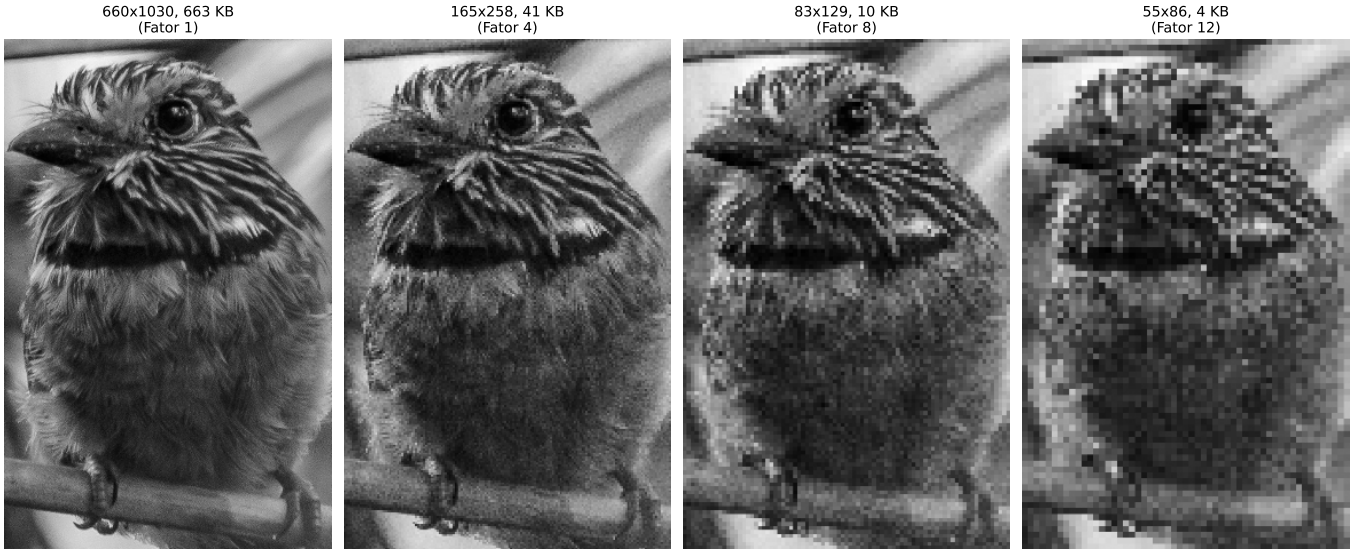


Figure 2.4: Effect of subsampling. The titles display the dimensions (W x H) and the size of the array in memory (KB).

The experiment in Figure 2.5 focuses on **intensity quantization**, the process of discretizing the amplitude of the function $f(x, y)$. While subsampling affects the spatial grid, reducing the bit depth limits the number of gray levels available to represent brightness.

By reducing the depth from 8 bits (256 levels) to smaller values, the **posterization** effect arises, where smooth gradients in a scene are replaced by abrupt transitions. At the 1-bit limit, the image becomes strictly binary, preserving only the silhouette and losing texture and volume details.

```
def quantize_simple(image, bits):
    """Reduces bit depth and computes memory metadata."""
    levels = 2 ** bits
    # Normalization and uniform quantization
    quantized = (np.floor(image / 256 * levels) / levels * 255).astype(np.uint8)

    # Memory calculation in KB
    mem_kb = quantized.nbytes / 1024
    label = f"{bits} bits ({levels} níveis)\n{int(mem_kb)} KB"

    return quantized, label

# List of bits to test (8 is the default, 1 is binary)
bits_test = [8, 4, 2, 1]

# Generates results and separates into lists for 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))
```

Figure 2.5: Effect of reducing bit depth. The titles display the number of bits, levels, and size in memory (KB).

2.5.4 Technical Analysis

- **Domain vs. Codomain**: Note that the spatial resolution (matrix dimensions) remains constant at 512x512; only the codomain of the image function changes.
- **Memory Constancy**: Observe in the titles that the size in KB does not decrease. This occurs because NumPy stores each quantized pixel in an 8-bit container (`uint8`), regardless of whether the actual value is only 0 or 1.
- **Perception**: Visual degradation becomes critical below 4 bits, where the human eye begins to perceive the artificial "boundaries" created by the lack of intermediate tones.

The limitation of data types smaller than a byte in the Python/NumPy ecosystem stems from hardware architecture, which addresses memory in blocks of 8 bits (Bytes). To maintain compatibility with OpenCV and ensure efficiency, even binary elements are mapped to 1-byte containers (`uint8` or `bool8`).

Although languages such as ANSI C allow the packing of 8 pixels per byte (*bit-packing*), this approach requires constant unpacking for computations and imposes high complexity in pointer manipulation. As per Table 2.2, the use of `uint8` is favored for ease of neighbor access and versatility in geometric transformations. Furthermore, native NumPy and OpenCV methods execute processing internally at a low level (C/C++), making vectorized operations faster than manual implementations with nested loops in Python.

Table 2.2: Comparison between packing strategies and processing efficiency.

Feature	Python (NumPy/OpenCV)	ANSI C (Bit-packing)
Smallest Unit	1 Byte (8 bits)	1 Bit
Memory (Binary)	256 KB (for 512x512)	32 KB (for 512x512)
Speed	High (C vectorization)	Variable (Slow if bit-shifting)
Complexity	Low: Ready-made methods	High: Pointers and Masks

2.6 Relationships Between Pixels - Image Topology

Pixels are not isolated elements; their relative positions define important concepts for processing.

2.6.1 Neighborhood

Given a pixel with coordinates (x, y), two main types of neighborhood are defined (for images on a rectangular *grid*):

- **4-Neighborhood** (von Neumann): includes the pixels at positions $(x-1, y)$, $(x+1, y)$, $(x, y-1)$, $(x, y+1)$.
- **8-Neighborhood** (Moore): includes all eight adjacent pixels (adds the four diagonal ones).

The choice of neighborhood influences operations such as edge detection, gradient computation, and connectivity.

```
# # Creation of a 3x3 matrix for topological example
# viz = np.zeros((3, 3), dtype='uint8')

# # Defining Neighborhood-4 (N4) with a different value for emphasis
# viz[0, 1] = viz[2, 1] = viz[1, 0] = viz[1, 2] = viz[1, 1] = 255

# or simply (test with numbers as arguments):
viz = mm.secross()

# Display of the matrix for coordinate analysis
mm.drawImgPlt(viz, scale=40)
```

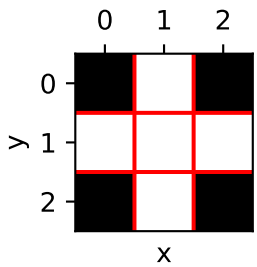


Figure 2.6: Illustration of 4-neighborhoods in a 3x3 matrix. At the center (1,1), the pixel of interest.

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

2.6.2 Adjacency, connectivity, and paths

Two pixels are **adjacent** if they are in contact according to a defined neighborhood **and** satisfy a value criterion (e.g., same intensity level). A **connectivity** defines an equivalence relation between pixels that form a connected region. A **path** is a sequence of adjacent pixels.

4-connectivity (N4) and **8-connectivity** (N8) can produce different results in segmentation and in the computation of connected components (labeling). For example, a checkerboard pattern can be completely disconnected under N4 but fully connected under N8.

2.6.3 Distances Between Pixels

Distance metrics are fundamental for quantifying physical proximity and connectivity among the elements that make up the digital grid. As demonstrated in Table 2.3, the choice of metric defines the movement cost between pixels and alters the behavior of segmentation algorithms and morphological analysis.

To measure the distance between two pixels $p(x_1, y_1)$ and $q(x_2, y_2)$, different metric functions are used, each imposing distinct movement constraints on the *grid*:

Table 2.3: Comparison of distance metrics applied to the pixel grid.

Metric	Definition	Interpretation
Euclidean	$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$	**Exact** straight-line distance (continuous)
Manhattan (*City block*)	$\lvert x_1 - x_2 \rvert + \lvert y_1 - y_2 \rvert$	Horizontal + vertical movements
Chebyshev (*Chessboard*)	$\max(\lvert x_1 - x_2 \rvert, \lvert y_1 - y_2 \rvert)$	Largest displacement among the axes

These distances are applied in various DIP contexts, including geometric interpolation algorithms, distance transforms, region growing, and shape analysis.

Practical Example

Considering two pixels with relative displacements $\Delta x = 3$ and $\Delta y = 4$:

- **Euclidean**: $\sqrt{3^2 + 4^2} = 5$ (hypotenuse of the right triangle).
- ***Manhattan***: $3 + 4 = 7$ (sum of the legs).
- ***Chebyshev***: $\max(3, 4) = 4$ (dominance of the largest displacement).

2.7 Image Storage

The choice of file format is a decisive step in the processing pipeline, as it determines how sampling and quantization data will be preserved or discarded. As presented in Table 2.4, each extension balances data fidelity and storage efficiency in a distinct manner.

Table 2.4: Main digital image storage formats and their applications in DIP.

Format	Characteristics	Typical Use
PGM	Simple grayscale map format (text or binary).	Academic research and Unix tools.
BMP	Uncompressed (or simple compression).	Windows, legacy applications.
PNG	**Lossless** compression.	Web, images with transparency.
JPEG	**Lossy** compression, ideal for photographs.	Photos, digital cameras.
TIFF	Supports multiple layers and varied compression.	Publishing, archiving.
RAW	Raw sensor data, unprocessed.	Professional photography.
DICOM	Medical standard with embedded clinical metadata (patient, equipment, protocol).	Radiology, tomography, magnetic resonance imaging.

Image metadata includes parameters such as width, height, bit depth, and color encoding. In scientific formats, calibration information and capture details are also preserved. When using the `mm.read()` function, the `morph` library automatically preserves this data so that the original properties of the image are respected.

In scientific and hospital contexts, the **DICOM** (*Digital Imaging and Communications in Medicine*) standard is favored to ensure that there is no loss of diagnostic precision. Public repositories such as **The Cancer Imaging Archive (TCIA)**, the **Alzheimer's Disease Neuroimaging Initiative (ADNI)**, and **PhysioNet** provide vast datasets in this format, including anonymized clinical metadata that is essential for scientific research.

2.7.1 Example: Metadata Extraction and GPS Location

Unlike the pure pixel matrix obtained through conventional reading—as in the bird image presented at the beginning of this chapter—the use of the `pil=True` argument in the `mm.read()` method changes the nature of the returned object (see Figure 2.7). While the default (`pil=False`) returns an RGB `numpy.ndarray`, reading with `pil=True` returns a specialized object from the Pillow library, capable of interpreting the **EXIF** header.

The **EXIF** header (*Exchangeable Image File Format*) functions as a technical repository of the capture, allowing the Pillow object to interpret a vast range of information that goes far beyond GPS coordinates. By using `pil=True`, the system gains access to the "DNA" of the image, including hardware metadata (camera brand and model), optical settings (aperture, focal length, and exposure time), and lighting parameters (flash usage and white balance).

This distinction is important in digital image processing, as it transforms the sampling matrix into a contextualized dataset, where the physical characteristics of the sensor and lens can be used to normalize brightness or correct geometric distortions.

```
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. Reading - preserves PIL object with 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)      # saving preserving EXIF
else:
    img_obj = mm.read(caminho, pil=True)

img_numpy = np.array(img_obj)      # conversion to NumPy

# 2. GPS extraction and conversion (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. Diagnosis of types, dimensions and pixel access
print(f"\nPIL Type  : {type(img_obj)}   | Dimensions (x,y): {img_obj.size}")
print(f"Pillow (0,0): {img_obj.getpixel((0, 0))}")
print(f"NumPy Type: {type(img_numpy)} | Dimensions [y,x,c]: {img_numpy.shape}")
print(f"NumPy [0,0]: {img_numpy[0, 0]}")

# 4. Display
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
```

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

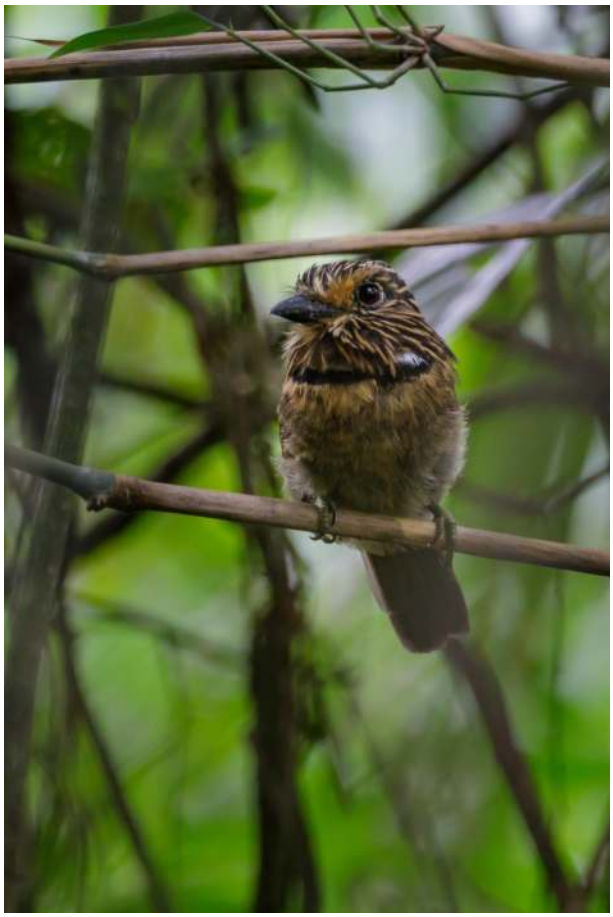


Figure 2.7: Environmental Protection Area Quilombos of Médio Ribeira - Barbudo-rajado (Malacoptila striata). Credit: Thomas Fuhrmann (CC BY-SA 4.0).

Pedagogical Note: The Subtle Difference in Dimensions

Notice that the representation of dimensions changes depending on the data structure used:

- **In Pillow (`.size`):** Returns `(Width, Height)` — in the example: `(2047, 3067)`. This is a view oriented toward the image file.
- **In NumPy (`.shape`):** Follows the mathematical matrix convention: `(Rows/Height, Columns/Width, Channels)` — in the example: `(3067, 2047, 3)`.

This distinction is fundamental to avoid indexing errors when implementing manual filters. While the Pillow object carries the **"where"** and **"when"** (context), the NumPy array carries the **"how much"** light (intensity) at each point of the image.

2.7.2 Why Is This Separation Important?

When loading an image through the conventional path (`pil=False`), the result is a `numpy.ndarray`, which strictly contains the numerical values resulting from **quantization** and **sampling**. However, when using `pil=True`, the `mm.read()` returns an object of class `PIL.JpegImagePlugin.JpegImageFile`.

This class keeps the file "open" to allow access to the capture context before the data are converted into a

raw matrix. Note that the pixel at `(0, 0)` is identical in both representations — `(58, 96, 0)` in Pillow and `[58, 96, 0]` in NumPy — confirming that both describe the same data, only with different interfaces. This separation is vital: pixels serve algorithms; metadata serve georeferencing, scientific cataloging, and corrections based on the acquisition hardware.

To inspect all EXIF metadata of a Pillow object:

```
from PIL import ExifTags

exif_raw = img_obj._getexif()
if exif_raw:
    for tag_id, valor in sorted(exif_raw.items()):
        tag_nome = ExifTags.TAGS.get(tag_id, f"TAG_{tag_id}")
        print(f"  {tag_nome:40s} : {valor}")
```

2.8 Basic Geometric Transformations

Geometric transformations alter the position of pixels while maintaining intensity values. They are fundamental for alignment, distortion correction, and data augmentation in machine learning.

An **affine transformation** is any mapping that preserves collinearity (points on a line remain on a line) and ratios of distances between collinear points. In 2D, every affine transformation can be expressed in **homogeneous coordinates** by a 3×3 matrix:

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

The upper-left 2×2 submatrix $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ controls rotation, scaling, and shearing; the vector $(t_x, t_y)^\top$ controls translation. The most commonly used geometric transformations in DIP — translation, rotation, and scaling — are special cases of $\mathbf{T}$, and can be **composed** through matrix multiplication, in the order $\mathbf{T} = \mathbf{T}_n \cdots \mathbf{T}_2 \mathbf{T}_1$.

> **Inverse transformation and interpolation**
>
> In practical implementation (`cv2.warpAffine`), the **inverse transformation** is applied: for each pixel (x', y') of the destination image, the source position $(x, y) = \mathbf{T}^{-1}(x', y')$ is computed and the value is interpolated. This avoids holes in the resulting image caused by directly mapping integer pixels to non-integer positions.

2.8.1 Translation

Translation is the simplest affine transformation: it shifts all pixels by a vector (t_x, t_y). In homogeneous coordinates, it is expressed by the matrix:

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

The third row of the matrix ensures that the operation remains in affine space, allowing translation, rotation, and scaling to be combined by simple matrix multiplication. In practice, `cv2.warpAffine` uses only the first two rows (a 2×3 matrix), since the third row is always $[0, 0, 1]$.

Pixels shifted beyond the original area are discarded; uncovered areas are filled with 0 (black). See an example in Figure 2.8.

```
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", "Translation (50,50)", "Translation (100,50)"],
        cols=3, figsize=(16, 12))
```

Figure 2.8: Example of translation of the bird image with offsets (50,50) and (100,50).

2.8.2 Rotation

Rotation by angle θ around a central point (c_x, c_y) is composed of three affine transformations: translation to the origin, pure rotation, and translation back. The resulting matrix is:

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

In the implementation, `cv2.getRotationMatrix2D` directly generates the first two rows of $\mathbf{T}_{\text{rot}}$ (a 2×3 matrix for `warpAffine`), also accepting a scale factor s that multiplies $\cos\theta$ and $\sin\theta$. See the example in Figure 2.9.

Since rotation moves pixels to new non-integer positions, `cv2.warpAffine` must estimate the color of each destination pixel from its neighbors — a process called **interpolation**. The `interp` parameter controls this behavior:

- **`nearest`** (`INTER_NEAREST`): assigns the value of the nearest pixel. Fast, but produces jagged edges (*aliasing*) on diagonal borders.
- **`bilinear`** (`INTER_LINEAR`, default): weighted average of the 4 nearest neighbors. Balances quality and performance — suitable for most cases.
- **`bicubic`** (`INTER_CUBIC`): considers the 16 neighbors on a cubic surface. Produces smoother edges at the cost of greater processing.

```
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", "Rotation 30°", "Rotation 45°"],
        cols=3, figsize=(16, 12))
```

Figure 2.9: Example of rotating the bird image by 30° and 45° using bilinear interpolation.

2.8.3 Scaling (Resizing)

Resizing by factors (s_x, s_y) is an affine transformation with matrix:

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

When $s > 1$ (enlargement), pixels of the destination image map to non-integer positions in the source — requiring interpolation to estimate the value. When $s < 1$ (reduction), multiple source pixels contribute to a single destination pixel — requiring **decimation**. The same three methods described for rotation are available in `mm.resize`, with the difference that here the visual impact is more noticeable: in enlargement, `nearest` produces a blocky effect (*pixelation*), whereas `bicubic` better preserves edge sharpness, as shown in Table 2.5:

Table 2.5: Interpolation methods available in mm.resize and their respective numbers of neighbors used in the calculation.

Method	Neighbors used	Characteristic
`'nearest'`	1	Fast; produces a blocky effect (*pixelation*)
`'bilinear'`	4	Good quality/cost trade-off; smooth edges
`'bicubic'`	16	Greater sharpness; preferred in professional software

The `size_or_factor` parameter accepts either a scalar (uniform factor, e.g. `0.5` to reduce to half) or a tuple `(width, height)` for absolute dimensions.

```
# 1. Crop the beak region
y, x, offset = 210, 40, 40
crop = img_gray[y-offset:y+offset, x-offset:x+offset]
print(f"Image: {img_gray.shape} | Crop: {crop.shape}")

# 2. Enlarge 4× with mm.resize
crop_nearest  = mm.resize(crop, size_or_factor=4, method='nearest')
crop_bilinear = mm.resize(crop, size_or_factor=4, method='bilinear')

# 3. Comparative display
mm.show(
    [crop, crop_nearest, crop_bilinear],
    titles=["Original (crop)", "Nearest neighbor (4×)", "Bilinear (4×)"],
    cols=3, figsize=(16, 12), dpi=200
)
```

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

Figure 2.10: Comparison of interpolation with zoom on the eye detail (60×60 crop, enlarged 4×). Note the blocky effect in nearest neighbor vs. the smoothing in bilinear.

2.8.4 Shearing

Shearing is an affine transformation that distorts the image by shifting each pixel proportionally to its position along one axis. The general matrix combines horizontal shear (sh_x) and vertical shear (sh_y):

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

For $sh_x \neq 0$ and $sh_y = 0$, each row is shifted horizontally proportionally to its vertical position — producing the characteristic "tilt" effect. See the example in Figure 2.11.

```
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)", "Combined (0.2, 0.2)"],
    cols=4, figsize=(16, 12)
)
```

Figure 2.11: Example of bird image shearing: horizontal (shx=0.3), vertical (shy=0.3) and combined (shx=0.2, shy=0.2).

2.9 Summary

This chapter presented the fundamentals of image digitization and topology:

- **Image formation:** $f(x, y) = i(x, y) \cdot r(x, y)$.
- **Sampling:** discretization of space $\rightarrow$ spatial resolution $M \times N$.
- **Quantization:** discretization of intensity $\rightarrow$ bit depth b.
- **Topological relations:** 4-neighborhood, 8-neighborhood, connectivity, distances (Euclidean, Manhattan, Chebyshev).
- **Geometric transformations:** translation, rotation, scaling (with bilinear or nearest-neighbor interpolation).
- **File formats:** BMP, PNG, JPEG, TIFF, RAW; each with different trade-offs between quality and size.

Chapter 3 will address **spatial operations** such as convolution, filtering, and mathematical morphology (erosion, dilation).

2.10 🤖 Using Gemini Notebook as a Complementary Tutor

In this edition, we encourage the use of **Gemini Notebook** as a complementary learning tool. This AI tool relies exclusively on the documents provided by the author as its knowledge base, ensuring responses consistent with the book's content.

For each chapter, we have prepared a specific project on the platform. For an enhanced study experience, use the access link below:

🎓 Study with the Intelligent Tutor

To interact with the content of this chapter, access the link below. The environment contains teaching materials in different formats, generated from the chapter's **PDF**. On the platform, especially explore the **Study Guide** and **Conversation** options to deepen your understanding.
🚀 ACCESS GEMINI NOTEBOOK: CHAPTER 02

🌐 **Language and Programming Language**

The project for this chapter in Gemini Notebook was built using only the text in **Portuguese** and code examples in **Python**. If you are studying from the English or French edition, or following the C++ track, the tutor's responses may not exactly match the version you are reading.

⚠️ **Notice on AI-Generated Content**

AI is a powerful ally in studying, but the generated content may contain **errors or inaccuracies**. Always consult **books, scientific articles, and other reliable academic sources** to validate the information. Whenever possible, run the practical examples provided in this chapter to verify the results.

2.11 Exercise List

1. **(15%)** Explain, in your own words, the difference between **sampling** and **quantization**. Give a concrete example of each in the context of a digital image.
2. **(15%)** Consider an image with a spatial resolution of 1024×768 pixels and a bit depth of 24 bits (8 bits per RGB channel). Calculate the total uncompressed size of the image in bytes and in megabytes.
3. **(20%)** Using the laboratory code, modify the subsampling factor to 3 and to 6. Visually describe what happens to the edges of objects. What is the *aliasing* effect?
4. **(20%)** For the grayscale image, apply quantization with 3 bits (8 levels) and 5 bits (32 levels). Compare the results and explain why 5 bits can already be considered sufficient for many applications.
5. **(15%)** Given two pixels $A = (10, 20)$ and $B = (15, 25)$, calculate the Euclidean, Manhattan, and Chebyshev distances between them.
6. **(15%)** Using the `mm.rotate` function, rotate the bird image at angles of 90°, 180°, and 270° with bilinear interpolation. Compare this with rotation using `method='nearest'`. In which situations is nearest-neighbor interpolation still useful?

Chapter References

The theoretical foundation of this chapter is based on the following works:

- Gonzalez (2018) for the concepts of sampling, quantization, and relationships between pixels.
- Szeliski (2022) for geometric transformations and connectivity.
- Bradski (2008) for the practical implementation with OpenCV and `morph.py`.

Run Colab | Open GitHub

2.12 💻 Practical Part with Programming Exercises

🎯 Objective of this Notebook

The notebook allows you to develop, validate, organize, and test solutions for **Programming Exercises (PEs)** in interactive environments, such as Colab, with the same test cases as Moodle, copying them there only when it is time to record the official grade.

Download

Download `morph.py` and `testsuite.py` by running the cell below:

```
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup(testsuite=True)
from morph import mm
from testsuite import TestSuite
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Executing Tests

To evaluate the tests, run `TestSuite("EP04_01.extension").run()` in a new cell, replacing the extension with that of the language used (`.py`, `.java`, `.c`, `.cpp`, `.js`, or `.r`). The system downloads the test cases from GitHub, runs the program, and computes the grade automatically.

To test Python code directly, without saving a file, use `run_code(code)` by passing the code as a *string* in a variable `code`:

```
code = """
from morph import mm
# ... your code here ...
"""
TestSuite("EP04_01").run_code(code)
```

2.12.1 EP02_01 ⊙ Brightness and Contrast Adjustment

In this activity, the goal is to implement a point operator for **linear intensity transformation**, applying dynamic brightness and contrast adjustment to a digital image.

2.12.1.1 📋 Implementation Guidelines

The algorithm should follow the execution flow below:

1. **Dimensions:** Read the integers L (rows) and C (columns) of the matrix.
2. **Parameters:** Read the real value α (contrast factor) and the integer β (brightness factor).
3. **Data:** Read the integer values of the original matrix.
4. **Mapping:** For each pixel p, compute the new value p' using the equation:

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

5. **Output:** Display the resulting matrix with dimensions $L \times C$.

2.12.1.2 📌 Computational Constraints

- **Rounding (*Round*):** Mathematical rounding to the nearest integer is applied before type conversion.
- **Clipping (*Saturation*):** Values must be confined to the range $[0, 255]$ to preserve the 8-bit standard:

$$\text{clip}(x) = \max(0, \min(255, x))$$

- **Simulation:** The effect of the parameters α and β on histogram correction can be observed in Figure 2.12.

2.12.1.3 🧠 Theoretical Background

These changes modify the image histogram to adjust the illumination profile and tonal distinction.

Parameter	Function	Visual Impact
α (***Alpha***)	Scalar	Modulates **Contrast**. If $\alpha > 1$, it expands the histogram; if $0 \leq \alpha < 1$, it compresses it.
β (***Beta***)	Additive	Modulates **Brightness**. If positive, it shifts the histogram to the right; if negative, to the left.
clip	Limiter	Restricts the dynamic range, preventing underflow and overflow errors.

2.12.1.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Values of `alpha` (α) and `beta` (β).
- Following lines: Numeric elements of the original matrix.

Output:

- Transformed matrix structured into L rows and C columns.

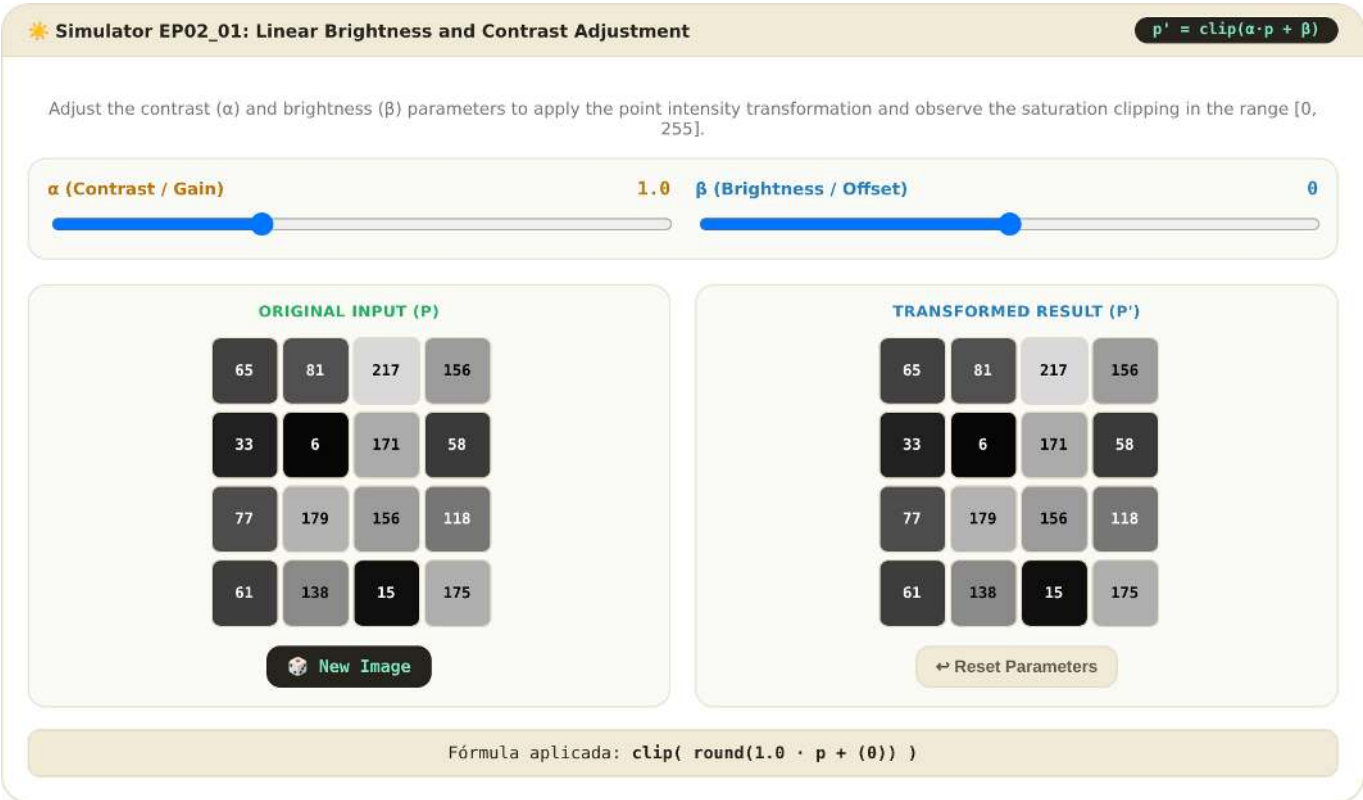
2.12.1.5 📌 Examples

The following table presents a practical example of the algorithm's expected behavior, highlighting the action of the rounding and clipping operators.

Input	Output	Observation
1 4 1.5 -30 0 100 180 255	0 120 240 255	Note the clipping effect on the last pixel

Running the Tests

To evaluate the tests, run `TestSuite("EP02_01.extensão").run()` in a new cell, replacing the extension with that of the language used (`.py`, `.java`, `.c`, `.cpp`, `.js`, or `.r`). The system downloads the test cases from GitHub, runs the program, and calculates the grade automatically.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap02/sim-ep0201-brilho-contraste.html

Figure 2.12: Simulator EP02_01: Linear Brightness and Contrast Adjustment (p' = p +)

```
%%writefile EP02_01.py
# your solution
```

```
Overwriting EP02_01.py
```

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

```
<IPython.core.display.HTML object>
```

2.12.2 EP02_02 🔬 Spatial Subsampling

In this activity, you must implement the reduction of the spatial resolution of an image through the subsampling process.

- Read two integers **L** and **C**, representing the dimensions of the original matrix.
- Read an integer value f ($f \geq 1$), which represents the sampling factor.
- Read the integer values of the original matrix.
- The new image must be constructed by selecting the pixel at position $(f \cdot i, f \cdot j)$ of the original image.
- Print the resulting matrix with the new dimensions.
- See a simulation of this EP in Figure 2.13.

📌 **Important**:

- **Final Dimensions:** The sampled image will have dimensions $\lceil L/f \rceil \times \lceil C/f \rceil$. In programming terms, this is equivalent to the resulting size of a slice with step f.
- **Implementation:** Do not use ready-made functions from image processing libraries (such as OpenCV or PIL) for resizing. Implement the pixel selection logic manually or via matrix slicing.
- **Aliasing:** Note that this process may cause the *aliasing* effect (jagged edges), where fine details are lost or unwanted patterns appear.

2.12.2.1 🧠 Discretization of Space

Subsampling reduces the spatial resolution of an image by selecting only one pixel every f pixels in each direction. It is the inverse process of interpolation:

Parameter	Function	Effect
Factor f	Sampling step	Defines the selection interval. A factor of 2 reduces the width and height by half.
Resolution	Pixel density	Decreases the total amount of spatial information in the image.
Aliasing	Side effect	Emergence of staircase patterns or blocks due to the loss of fine details.

2.12.2.2 📋 Task (specification for VPL)

Input:

The first line contains `L`.

The second line contains `C`.

The third line contains the factor `f`.

The following lines contain the elements of the $L \times C$ matrix.

Output:

The reduced matrix with dimensions corresponding to the slicing by `f`.

2.12.2.3 📌 Examples

Input	Output	Observation
2 4 2 10 20 30 40 50 60 70 80	10 30	Factor 2 selects pixels (0,0) and (0,2) from the first row. The second row is ignored.

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

```
Overwriting EP02_02.py
```

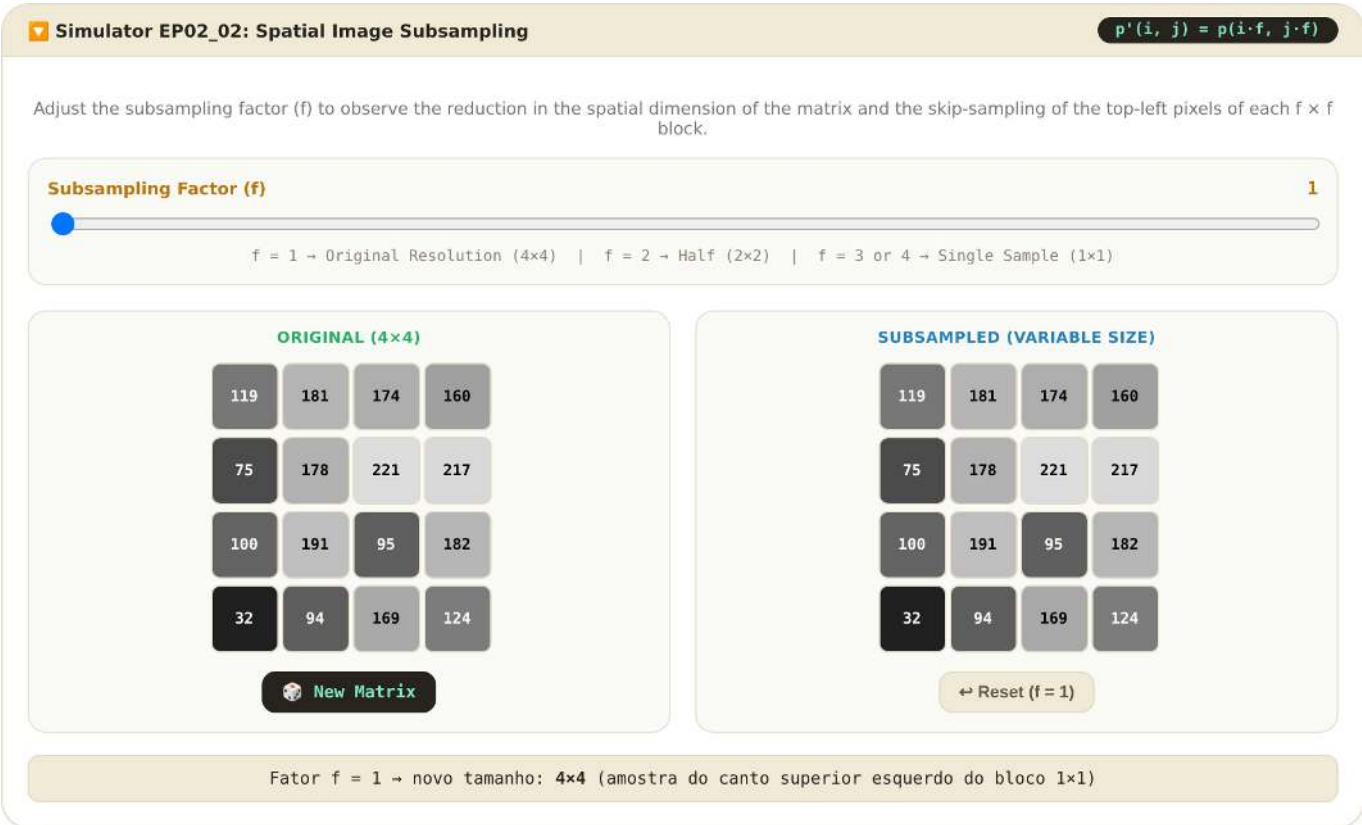
```
TestSuite("EP02_02.py").run()
```

```
<IPython.core.display.HTML object>
```

2.12.3 EP02_03 🎨 Gray Level Quantization

In this activity, you must implement uniform quantization of an image, reducing the number of original gray intensity levels to a new scale based on a smaller number of bits.

- Read two integers **L** and **C**, representing the dimensions of the matrix.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap02/sim-ep0202-subamostragem.html

Figure 2.13: EP02_02 Simulator: Spatial Subsampling (Resolution Reduction by f-Step)

- Read an integer k ($1 \leq k \leq 8$), representing the new number of bits of the image.
- Calculate the number of levels ($N = 2^k$) and the interval size (step).
- For each pixel p, calculate the new value p' by mapping it to the index of the corresponding discretized level (ranging from 0 to $2^k - 1$).
- Print the resulting matrix with the same original dimension values.
- See Figure 2.14 for a simulation of this EP.

📌 **Important**:

- **Posterization:** When drastically reducing the number of levels (e.g., $k = 2$), you will notice that smooth gradients become abrupt color bands due to the loss of amplitude resolution.
- **Step Calculation:** The interval between each level is defined by $step = 256/2^k$.
- **Mapping:** The uniform quantization method by truncation that maps the pixel to the index of its respective discretized level is given by:

$$p' = \left\lfloor \frac{p}{step} \right\rfloor$$

In terms of implementation (as in Python), this is equivalent to integer division: `p' = p // step`.

2.12.3.1 🧠 Amplitude Discretization

While subsampling deals with spatial resolution, quantization focuses on the precision of color (amplitude). Reducing bits means simplifying the chromatic information:

Parameter	Function	Effect
Bits (k)	Color depth	Defines how many different tones the image can have (2^k).
Step	Tone interval	Spacing between the allowed gray levels.
Posterization	Visual phenomenon	Transformation of continuous variations into blocks of solid color.

2.12.3.2 📋 Task (VPL specification)

Input:

The first line contains `L`.

The second line contains `C`.

The third line contains the number of bits `k`.

The following lines contain the elements of the $L \times C$ matrix.

Output:

The transformed matrix with the indices of the quantized levels, maintaining the original size $L \times C$.

2.12.3.3 📌 Examples

Input	Output	Observation
1 4 2 0 80 170 255	0 1 2 3	With $k = 2$, we have $2^2 = 4$ discrete levels available $(0, 1, 2, 3)$. The step is $256/4 = 64$. Applying integer division element-wise: $0//64 = 0$, $80//64 = 1$, $170//64 = 2$, $255//64 = 3$.
1 5 1 10 50 120 200 250	0 0 0 1 1	With $k = 1$, we have $2^1 = 2$ levels (0 and 1). Step $= 256/2 = 128$. Pixels less than 128 result in 0, and pixels greater than or equal to 128 result in 1.

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

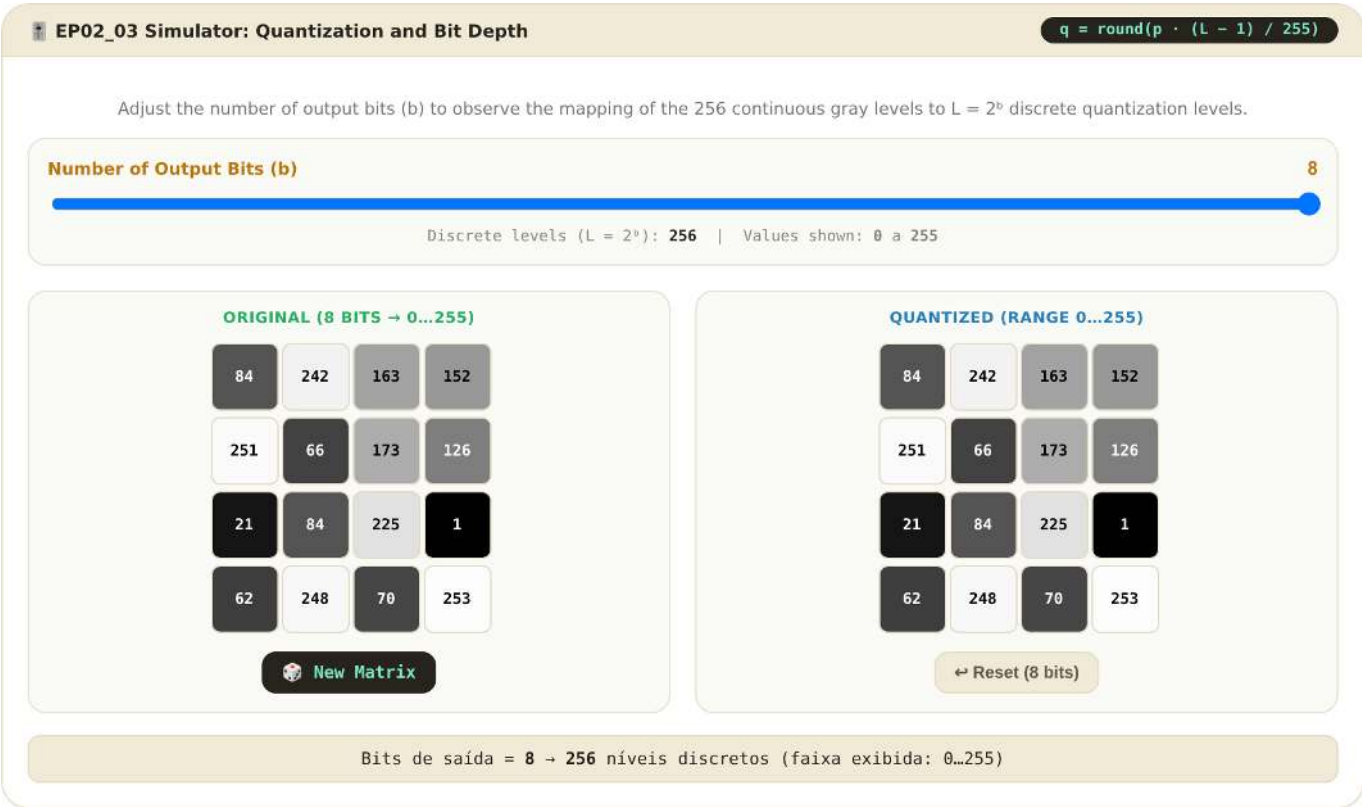
```
Overwriting EP02_03.py
```

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

```
<IPython.core.display.HTML object>
```

2.12.4 EP02_04 📐 Distance Transform in Binary Images

Given a binary image where pixels with value **1** represent the *object* and pixels with **0** represent the *background*, the distance of a background pixel is the **smallest distance to the nearest object pixel**. Object pixels



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap02/sim-ep0203-quantizacao.html

Figure 2.14: EP02_03 Simulator: Quantization and Bit Depth (Reduction of Gray Levels)

are assigned a distance of **0**. For simplicity, consider that the image has **only a single object consisting of one pixel with value 1**.

Problem: Read a binary image $L \times C$ and a metric, and compute this simplified distance by applying one of the three formulas:

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

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

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

where Δr is the row difference and Δc is the column difference between two pixels.

2.12.4.1 🖼️ Why does this matter? - Applications of the DT

The Distance Transform (DT) appears in dozens of computer vision pipelines:

Metric	Complexity	Typical application
Euclidean	🔴 $O(n^2)$ naive	Skeletonization, shape matching
City-block	🟡 $O(n)$ with 2 passes	Morphology, dilation/erosion
Chessboard	🟢 $O(n)$ with 2 passes	Morphology, dilation/erosion

2.12.4.2 📌 Technical Requirements

- **Input:** * First line: L and C (integers).
  - Second line: metric name (`euclidean`, `cityblock`, or `chessboard`).

  - Then, the binary matrix $L \times C$ (values 0 or 1).
- **Object pixels (1):** distance = 0 (or 0.00 for Euclidean).

- **Background pixels (0):** distance to the single object pixel in the image.

- **Rounding (Euclidean):** print with **2 decimal places** (format `:.2f`). City-block and Chessboard produce integers — print without decimals.

- **Output:** space-separated values, one row of the matrix per line.
- See Figure 2.15 for a simulation of this EP.

2.12.4.3 📌 Examples

Input	Output	Observation
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	The Chessboard distance is $\max(\lVert dx \rVert, \lVert dy \rVert)$. The only object pixel is $(2, 2) = 0$; the others store their minimum distance to it.

2.12.4.4 📌 Final Remarks

- Since the image has **only one single-pixel object**, the distance of each background pixel is simply the distance from that pixel to the single object point.
- The implementation may use brute force (iterate over all image pixels and compute the distance directly), since L and C are small in the test cases.
- This problem serves as a warm-up for the general Distance Transform, which will be addressed in later chapters with multiple objects and optimized algorithms.

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

```
Overwriting EP02_04.py
```

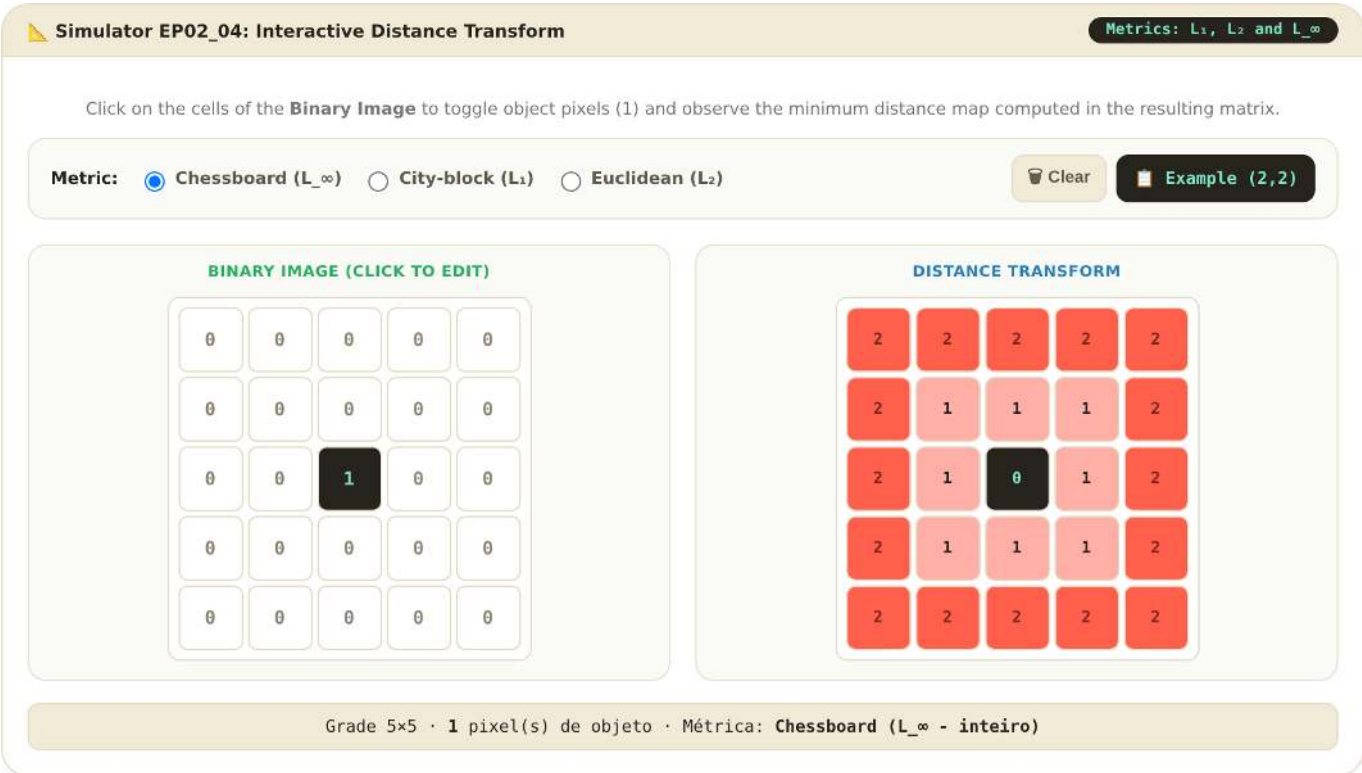
```
TestSuite("EP02_04.py").run()
```

```
<IPython.core.display.HTML object>
```

2.12.5 EP02_05 Image Translation

In this activity, you must implement the spatial displacement of an image. Translation moves each pixel of the original image to a new position based on a displacement vector.

- Read two integers **L** and **C**, representing the dimensions of the matrix.
- Read two integers t_x (horizontal displacement) and t_y (vertical displacement).
- Read the integer values of the original matrix.
- Calculate the new position (x', y') for each original pixel (x, y).



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap02/sim-ep0204-distancia.html

Figure 2.15: EP02_04 Simulator: Distance Transform in Binary Image (Chessboard, City-block and Euclidean)

- Print the resulting matrix with the same dimensions as the original.
- See a simulation of this EP in Figure 2.16.

📌 **Important**:

- **Filling:** Pixels that "enter" the image due to displacement and have no corresponding pixel in the original must be filled with **0** (black).
- **Discarding:** Pixels that, after translation, fall outside the matrix boundaries ($0 \dots L-1$ or $0 \dots C-1$) must be ignored.
- **Coordinates:** Consider x as the row index and y as the column index.

2.12.5.1 🧠 Spatial Displacement

Translating an image means moving all its points by a fixed distance in specified directions. Mathematically, using homogeneous coordinates, the operation is described as:

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

Which results in the simple equations:

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

2.12.5.2 📋 Task (specification for VPL)

Input:

The first line contains `L`.

The second line contains `C`.

The third line contains the integers `tx` and `ty`.

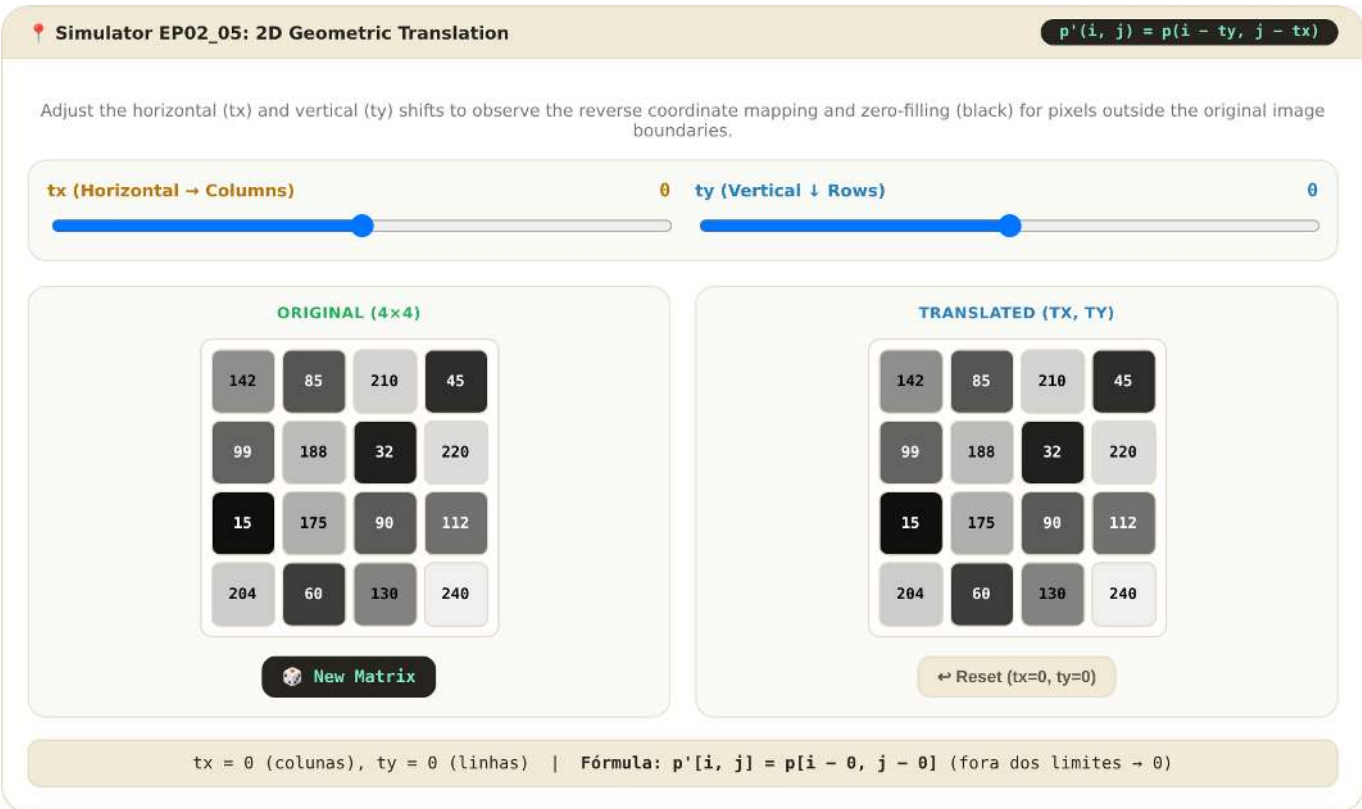
The following lines contain the elements of the $L \times C$ matrix.

Output:

The resulting matrix with the same dimensions $L \times C$ after displacement.

2.12.5.3 📌 Examples

Input	Output	Observation
2 2 1 1 10 20 30 40	0 0 0 10	**Displacement ($t_x = 1, t_y = 1$):** Each pixel moves one position to the right (horizontal) and one position down (vertical). Pixel $(0, 0) = 10$ moves to destination $(1, 1)$ (bottom-right corner). Empty positions are filled with 0.
3 3 -1 0 1 2 3 4 5 6 7 8 9	2 3 0 5 6 0 8 9 0	**Displacement ($t_x = -1, t_y = 0$):** Each pixel moves one position to the left (horizontal). The original first column (`1`, `4`, `7`) is discarded, the remaining columns move to the left, and the last resulting column is filled with zeros (0).



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap02/sim-ep0205-translacao.html

Figure 2.16: EP02_05 Simulator: Geometric Image Translation (Displacement tx and ty with Border Filling)

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

```
Overwriting EP02_05.py
```

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

```
<IPython.core.display.HTML object>
```

2.12.6 EP02_06 🔄 Image Rotation

In this activity, you must implement the rotation of an image around its geometric center. This operation requires coordinate mapping and the use of interpolation techniques to determine the new pixel values.

- Read two integers **L** and **C**, representing the dimensions of the matrix.
- Read a real value θ (angle in degrees) and a *string* representing the **interpolation** method (`nearest` or `bilinear`).
- Read the integer values of the original matrix.
- Perform the rotation around the image center $(L/2, C/2)$.
- Print the resulting matrix with the same dimensions as the original.
- See Figure 2.17 for a simulation of this EP.

📌 **Important**:

- **Inverse Mapping:** To avoid "holes" in the final image, iterate over each pixel (x', y') of the destination image and compute its corresponding position (x, y) in the original image using the inverse rotation matrix.
- **Interpolation:**
- `nearest`: Assigns the value of the pixel closest to the computed coordinate.
- `bilinear`: Computes a weighted average based on the 4 nearest neighbors.
- **Borders:** Pixels whose origin (x, y) falls outside the bounds of the original image must be filled with **0**.

2.12.6.1 🧠 Angle Transformation

The rotation of a point (x, y) relative to the origin by an angle θ is given by the transformation matrix. To rotate around a center (x_c, y_c), we first translate the center to the origin, rotate, and translate back:

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

Tip: Use inverse mapping to ensure that all pixels of the output image are correctly filled.

2.12.6.2 📋 Task (specification for VPL)

Input:

The first line contains `L`.

The second line contains `C`.

The third line contains the angle `theta` (in degrees) and the method `interp` (`nearest` or `bilinear`).

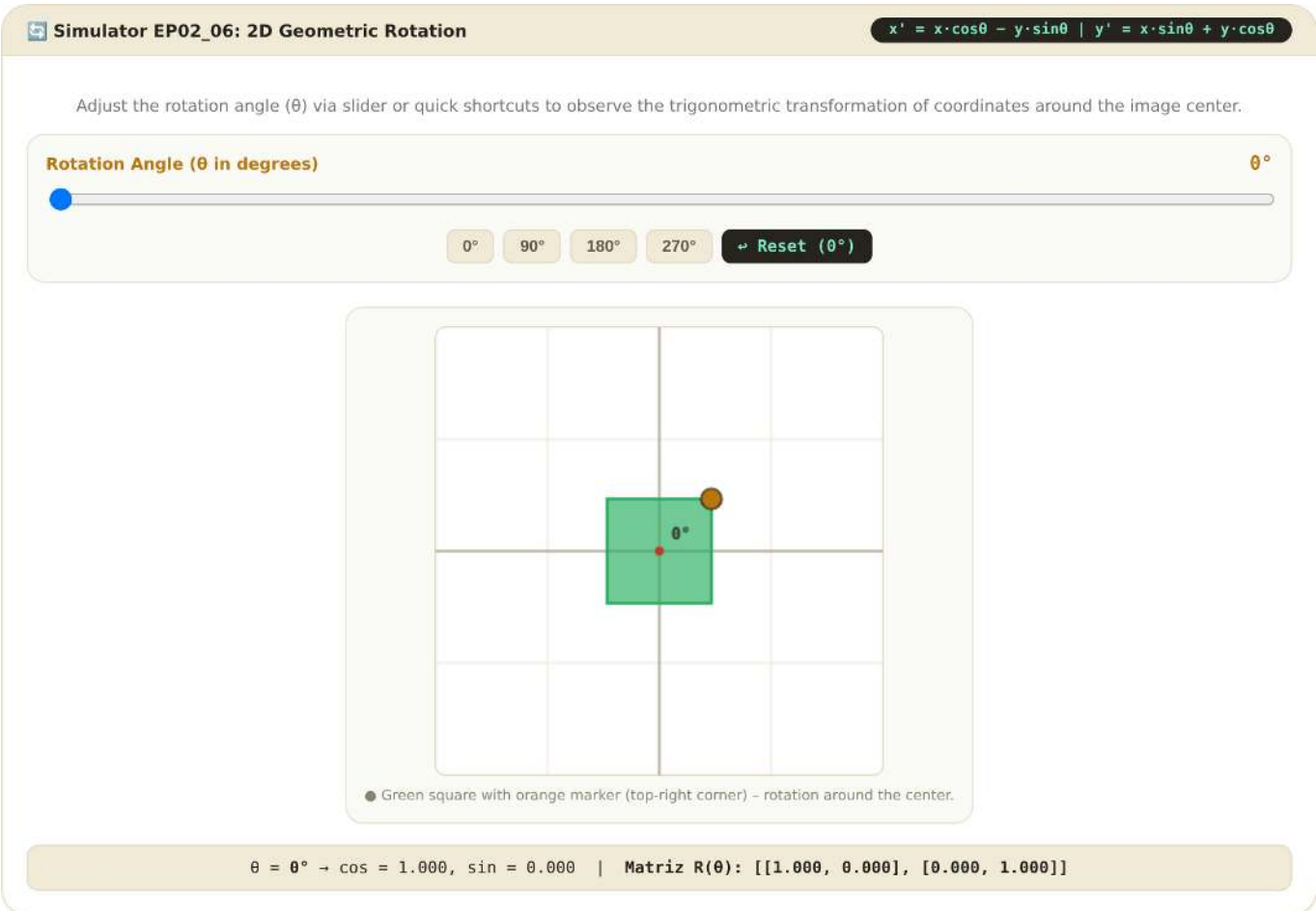
The following lines contain the elements of the $L \times C$ matrix.

Output:

The rotated matrix with `L` rows and `C` columns.

2.12.6.3 📌 Examples

Input	Output	Observation
2 2 90 nearest 1 2 3 4	3 1 4 2	90° clockwise rotation: column 0 becomes row 0 (from bottom to top). $(0,0) = 1 \to (1,0)$, $(1,0) = 3 \to (0,0)$, $(0,1) = 2 \to (1,1)$, $(1,1) = 4 \to (0,1)$.
3 3 45 bilinear 0 0 0 0 255 0 0 0 0	0 180 0 180 255 180 0 180 0	45° rotation: the central pixel remains 255; the direct neighbors receive an interpolated value ≈ 180 via bilinear; the corners remain 0.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap02/sim-ep0206-rotacao.html

Figure 2.17: EP02_06 Simulator: Image Rotation Around Origin by Angle

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

```
Overwriting EP02_06.py
```

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

```
<IPython.core.display.HTML object>
```

2.12.7 EP02_07 🔍 Resizing (Scaling)

In this activity, you must implement image resizing using scale factors. Unlike simple subsampling, here we will use interpolation techniques to allow both image enlargement and reduction.

- Read two integers **L** and **C**, representing the dimensions of the original matrix.
- Read two real values s_x (scale along rows) and s_y (scale along columns).
- Read a *string* representing the interpolation **method** (`nearest` or `bilinear`).
- Read the integer values of the original matrix.
- Compute the new dimensions: $L' = \text{round}(L \times s_x)$ and $C' = \text{round}(C \times s_y)$.
- Print the resulting matrix with the new dimensions.
- See Figure 2.18 for a simulation of this EP.

📌 **Important**:

- **Inverse Mapping:** For each pixel (x', y') of the destination image, find the corresponding position in the source using $(x, y) = (x'/s_x, y'/s_y)$.
- **Interpolation:**
- `nearest`: Selects the value of the nearest pixel (rounding the coordinates).
- `bilinear`: Performs double linear interpolation among the four nearest neighboring pixels in the original image.
- **Boundaries:** Ensure that the mapping does not attempt to access indices outside the range $[0, L-1]$ and $[0, C-1]$.

2.12.7.1 🧠 Interpolation for Enlargement/Reduction

Resizing an image by factors (s_x, s_y) requires filling gaps (in enlargement) or merging information (in reduction). The interpolation method defines the visual quality of the result:

Method	Operation	Visual Effect
Nearest	Takes the value of the nearest neighbor.	Fast, but produces a "pixelated" or blocky effect.
Bilinear	Weighted average of the 4 neighbors (2×2).	Smooths the image, reducing jaggedness.

2.12.7.2 📋 Task (specification for VPL)

Input:

The first line contains `L`.

The second line contains `C`.

The third line contains the factors `sx` and `sy`.

The fourth line contains the method `interp` (`nearest` or `bilinear`).

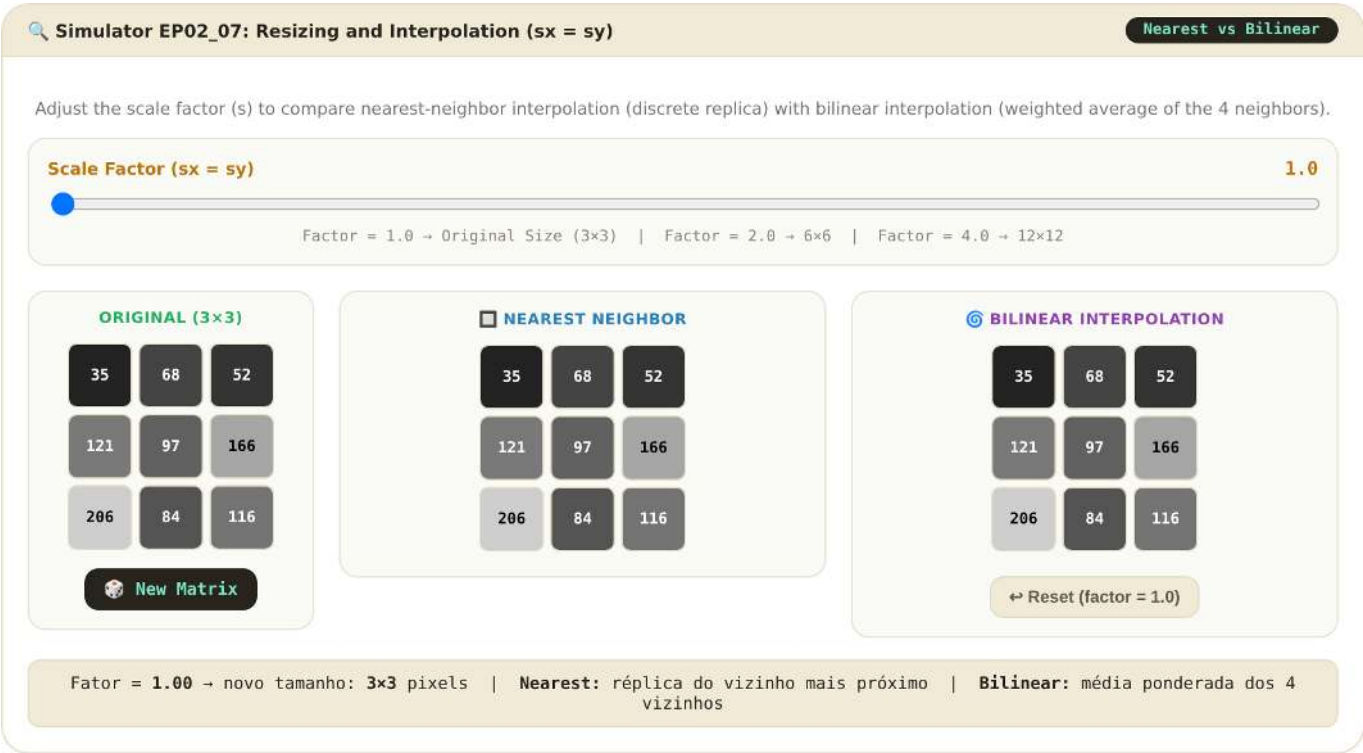
The following lines contain the elements of the $L \times C$ matrix.

Output:

The resized matrix with dimensions $L' \times C'$.

2.12.7.3 📌 Examples

Input	Output	Observation
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	2× enlargement: each original pixel is replicated in a 2×2 block. The 2×2 image becomes 4×4.
2 2 0.5 0.5 nearest 10 20 30 40	10	0.5× reduction: the 2×2 image becomes 1×1. With nearest, the only output pixel samples position $(0, 0) = 10$.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap02/sim-ep0207-escala.html

Figure 2.18: EP02_07 Simulator: Spatial Resizing and Interpolation (Nearest Neighbor vs Bilinear)

```
%%writefile EP02_07.py
# Python code
import numpy as np
from morph import mm

# 1. Reading dimensions, factors, and method
l = int(input())
c = int(input())
sx, sy = map(float, input().split())
interp = input().strip()

# 2. Reading the original image
img = mm.readImg(l, c)
```

```
# 3. New dimensions
l_new = round(l * sx)
c_new = round(c * sy)

# 4. Resizing using mm.resize
# cv2.resize uses (width, height) = (columns, rows)
resultado = mm.resize(img, (c_new, l_new), method=interp)

# 5. Display
print(mm.drawImg(resultado))
```

```
Overwriting EP02_07.py
```

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

```
<IPython.core.display.HTML object>
```

2.12.8 EP02_08 🔀 Shear Transformation

In this activity, you must implement the shear transformation on an image. Shear is an affine transformation that shifts each point in a fixed direction, by an amount proportional to its distance from a line parallel to that direction, resulting in a tilting effect.

- Read two integers **L** and **C**, representing the matrix dimensions.
- Read two real values sh_x (horizontal shear) and sh_y (vertical shear).
- Read a *string* representing the interpolation **method** (`nearest` or `bilinear`).
- Read the integer values of the original matrix.
- Apply the transformation while maintaining the original image size (cropping anything that exceeds the boundaries).
- Print the resulting matrix with dimensions $L \times C$.
- See Figure 2.19 for a simulation of this EP.

📌 **Important**:

- **Inverse Mapping:** For each pixel (x', y') of the destination image, compute the corresponding position in the source (x, y) using the inverse shear matrix.
- **Filling:** Coordinates that result in positions outside the original matrix must be filled with **0**.
- **Coordinates:** For the purposes of this implementation, consider x as the row index and y as the column index.

2.12.8.1 🧠 Affine Distortion

Shear alters the image geometry by tilting its axes. The relationship between the original coordinates (x, y) and the transformed ones (x', y') is given by:

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

This results in the following equations:

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

2.12.8.2 📋 Task (VPL specification)

Input:

The first line contains `L`.

The second line contains `C`.

The third line contains the factors `shx` and `shy`.

The fourth line contains the method `interp` (`nearest` or `bilinear`).

The following lines contain the elements of the $L \times C$ matrix.

Output:

The transformed matrix with the same dimensions $L \times C$.

2.12.8.3 📌 Examples

Input	Output	Observation
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	Horizontal shear: row i shifts by $\lfloor i \cdot 0.5 \rfloor$ pixels. Row $0 \to 0$px, row $1 \to 0$px, row $2 \to 1$px. Pixels shifted out are discarded, and empty positions are filled with 0.
2 2 0.0 1.0 nearest 10 20 30 40	10 0 30 20	Vertical shear: column j shifts down by $\lfloor j \cdot 1.0 \rfloor$ pixels. Column $0 \to 0$px (unchanged), column $1 \to 1$px: 20 moves down to $(1, 1)$ and $(0, 1)$ becomes 0.

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

```
Overwriting EP02_08.py
```

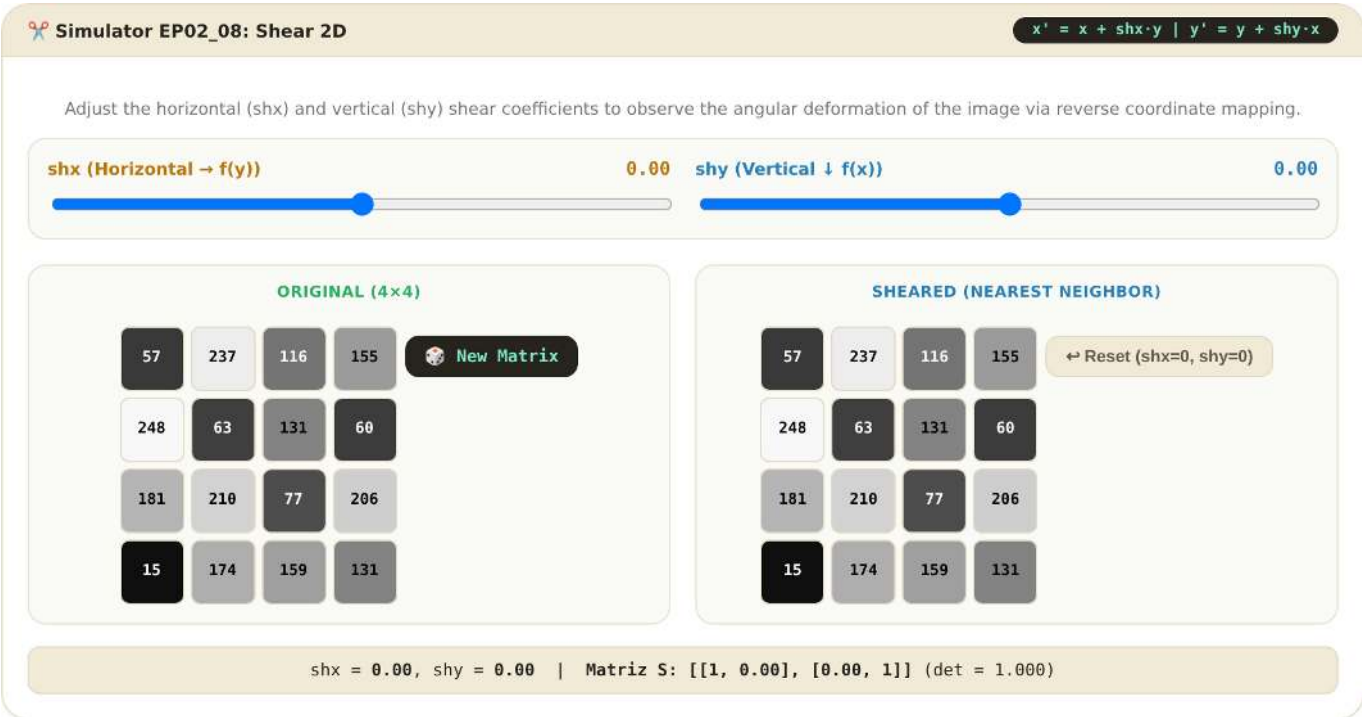
```
TestSuite("EP02_08.py").run()
```

```
<IPython.core.display.HTML object>
```

2.12.9 EP02_09 🧩 Generic Affine Transformation

In this activity, you must implement an arbitrary affine transformation on an image. This operation is the generalization of all linear transformations (scaling, rotation, shearing) combined with translation, allowing complex geometric manipulations through a single matrix.

- Read two integers **L** and **C**, representing the dimensions of the matrix.
- Read **six real values** (a, b, t_x, c, d, t_y) that compose the 2×3 affine transformation matrix.
- Read a *string* representing the **interpolation** method (`nearest` or `bilinear`).
- Read the integer values of the original matrix.
- Apply the transformation while maintaining the original size $L \times C$.
- Print the resulting matrix.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap02/sim-ep0208-cisalhamento.html

Figure 2.19: EP02_08 Simulator: 2D Shear Geometric Transformation (Horizontal and Vertical Shear)

- See Figure 2.20 for a simulation of this EP.

📌 **Important**:

- **Inverse Mapping:** To compute the value of each pixel in the destination image, you must use the **inverse** of the provided affine transformation matrix to find the corresponding coordinate in the original image.
- **Filling:** Computed coordinates that fall outside the bounds $[0, L-1]$ and $[0, C-1]$ of the original image must result in a pixel with value **0**.
- **Flexibility:** This implementation must be able to perform any of the previous tasks (translation, rotation, etc.) simply by changing the matrix parameters.

Hint:

```
flags = cv2.INTER_NEAREST if interp == 'nearest' else \
        cv2.INTER_CUBIC   if interp == 'bicubic'  else \
        cv2.INTER_LANCZOS4 if interp == 'lanczos' else \
        cv2.INTER_LINEAR

r = cv2.warpAffine(img, M, (C, L), flags=flags)
```

2.12.9.1 🧠 Combining Operations

The affine transformation preserves points, lines, and planes. In image processing, it maps the position (x, y) to (x', y') following the system:

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

Or, compactly in homogeneous coordinates:

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

2.12.9.2 📋 Task (specification for VPL)

Input:

The first line contains `L`.

The second line contains `C`.

The third line contains six floats: `a b tx c d ty`.

The fourth line contains the method `interp` (`nearest` or `bilinear`).

The following lines contain the elements of the $L \times C$ matrix.

Output:

The transformed matrix with the original dimensions $L \times C$.

2.12.9.3 📌 Examples

Input	Output	Observation
2 2 1.0 0.0 0.5 0.0 1.0 0.5 bilinear 10 20 30 40	15 20 25 30	Fractional translation ($t_x = 0.5, t_y = 0.5$): each output pixel (i, j) samples the position $(i + 0.5,\ j + 0.5)$ from the input via bilinear interpolation. E.g., $(0, 0)$ interpolates the four neighbors $\rightarrow$ 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	2× scaling via the affine matrix ($a = 2, d = 2$): each output pixel (i, j) samples the position $(2i, 2j)$ from the input with nearest. E.g., $(0, 2) \rightarrow (0, 4)$ outside the image $\rightarrow$ nearest clips to $(0, 2) = 3$… awaiting confirmation of the border logic.

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

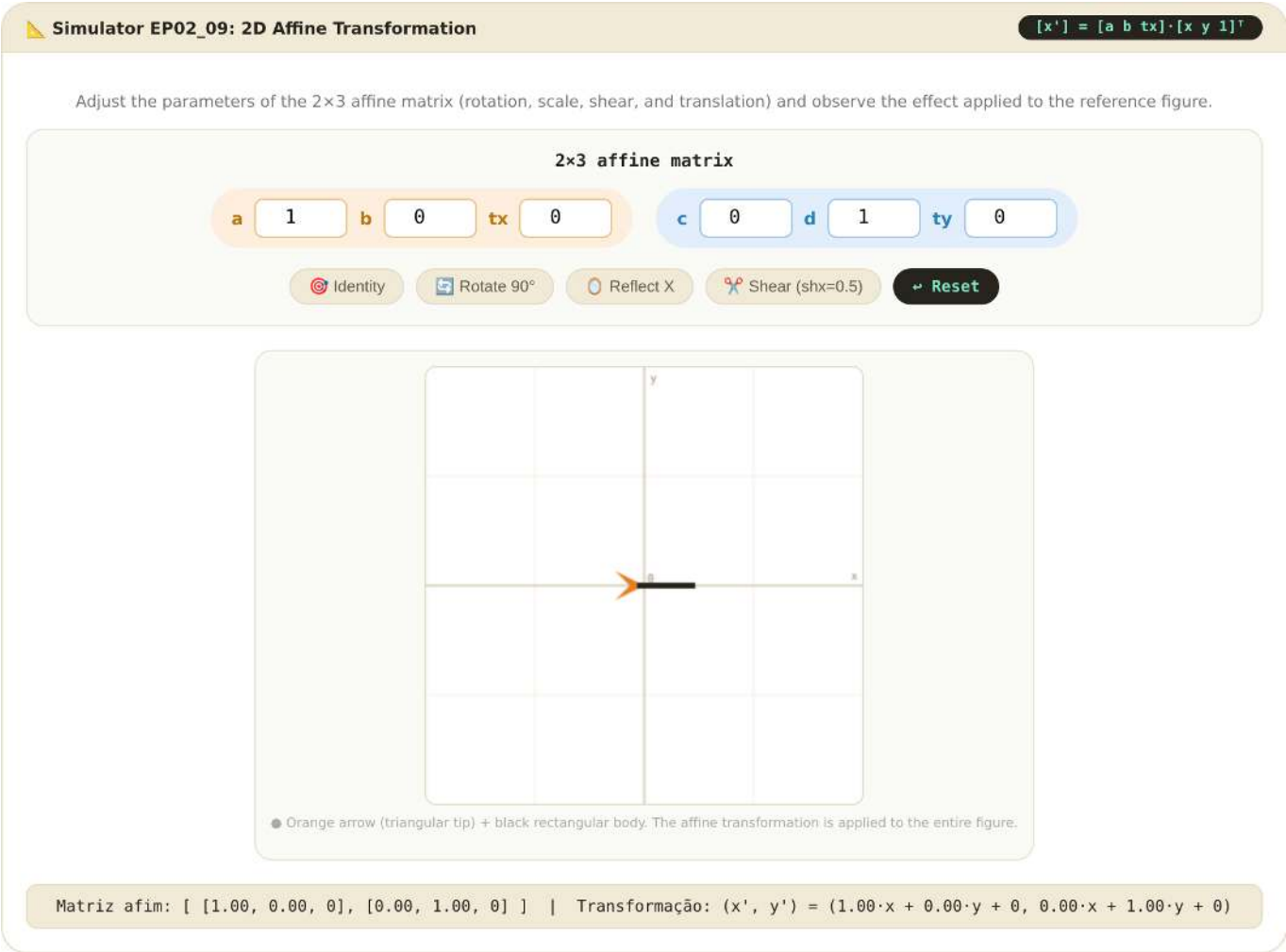
```
Overwriting EP02_09.py
```

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

```
<IPython.core.display.HTML object>
```

2.12.10 EP02_10 🎯 Perspective Correction (Homography)

In this activity, you must implement the perspective transformation, also known as homography. Unlike affine transformations, perspective does not preserve parallelism, allowing you to "rectify" tilted objects, such as documents or signs captured at oblique angles.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap02/sim-ep0209-afim.html

Figure 2.20: Simulator EP02_09: Affine Transformation 2D (2×3 Matrix)

- Read two integers **L** and **C**, representing the dimensions of the original matrix.
- Read **four coordinate pairs** (x, y) representing the corners of the source quadrilateral (distorted object).
- Read **four coordinate pairs** (x, y) representing the corners of the destination quadrilateral (where the object should be mapped).
- Read the values of the original matrix.
- Compute the 3×3 homography matrix and apply the transformation.
- Print the resulting matrix with the specified output dimensions.
- See Figure 2.21 for a simulation of this EP.

📌 **Important**:

- **Degrees of Freedom:** The homography has 8 degrees of freedom (the ninth element of the 3×3 matrix is a normalization constant, usually 1), requiring at least 4 corresponding points to be computed.
- **Projection:** After multiplying the coordinates by the matrix, you must divide the results x' and y' by the homogeneous component w to return to the 2D plane.
- **Use of Libraries:** For this task, you may use the functions `cv2.getPerspectiveTransform` to obtain the matrix and `cv2.warpPerspective` to apply the transformation, or implement the linear system and inverse mapping manually for an extra challenge.

```
# Output dimensions: bounding box of destination points + 1
w = int(max(pts2[:, 0])) + 1; h = int(max(pts2[:, 1])) + 1
# M = cv2.getPerspectiveTransform(pts1, pts2)
# dst = cv2.warpPerspective(img, M, (w, h))
# or
dst = mm.perspective_transform(img, pts1, pts2, size=(w, h))
```

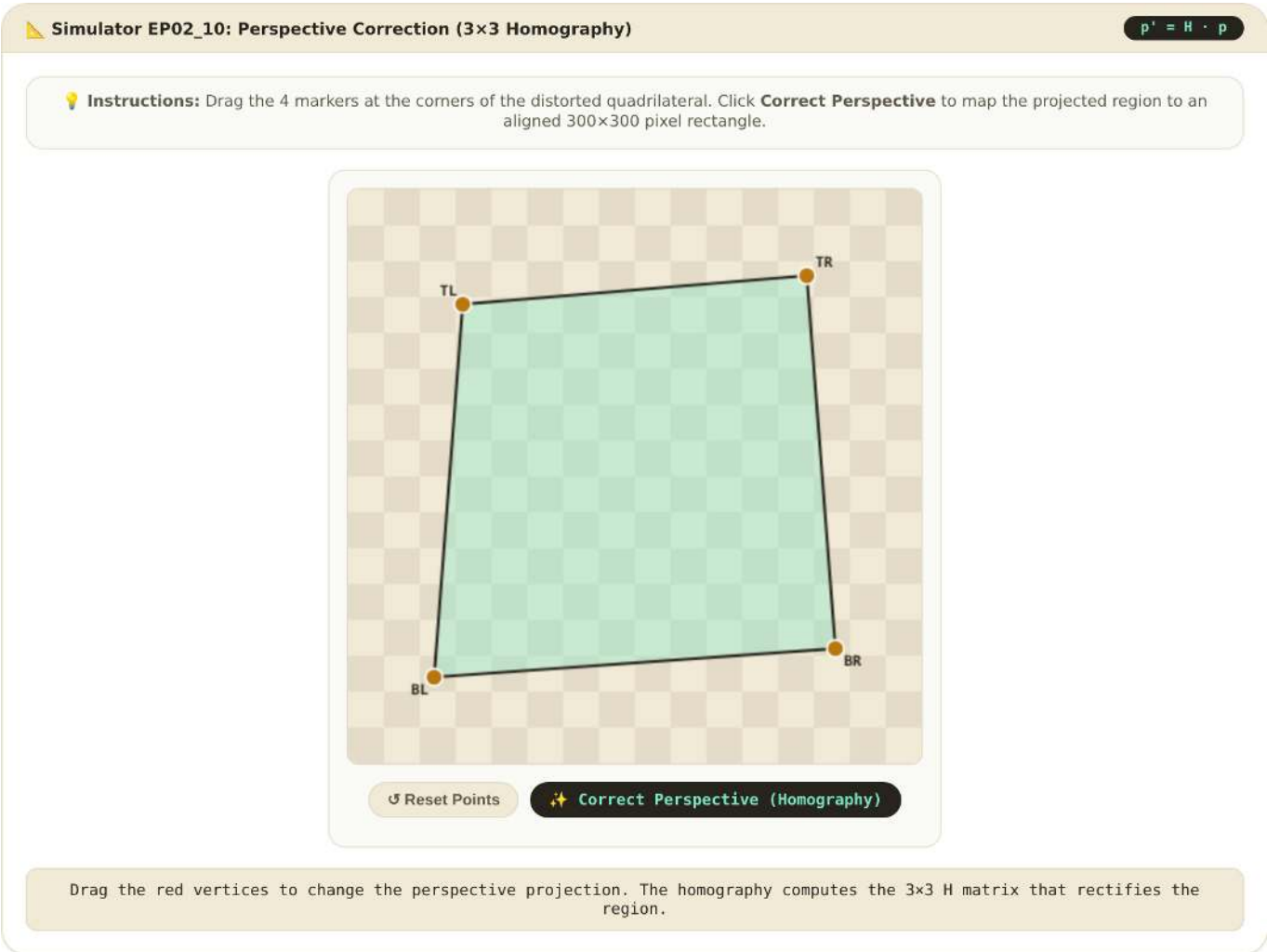
2.12.10.1 🧠 Non-affine Deformation

While affine transformations map parallelograms to parallelograms, the homography maps any quadrilateral to another quadrilateral. This is essential for computer vision:

Operation	Characteristic	Typical Application
Homography	Plane projection	Document correction, plate scanning.
Vanishing Point	Line convergence	3D reconstruction from 2D images.
Warping	Mesh deformation	Video stabilization and panoramas (stitching).

2.12.10.2 📌 Examples

Input	Output	Observation
4 4 0 0 3 0 0 3 3 3 0 0 3 0 0 3 3 3 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160	10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160	The first 4 lines after the dimensions are the source points; the following 4 are the destinations. With identical points, the perspective transformation is the identity and the image is preserved.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap02/sim-ep0210-perspectiva.html

Figure 2.21: Simulator EP02_10: Perspective Correction (3×3 Homography Transformation)

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

```
Overwriting EP02_10.py
```

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

```
<IPython.core.display.HTML object>
```

2.12.11 EP02_11 🏆 Perspective Correction (Homography) in a Real Image

In this activity, the goal is to apply perspective transformation (homography) to "rectify" a tilted object in a real photograph. You will work with an image of a newspaper, where the grid of a Sudoku puzzle is distorted due to the angle at which the photo was taken.

Your program must read input parameters from the terminal, load the image, compute the 3×3 homography matrix, apply the geometric transformation, and display a global validation indicator.

- Read two integers **L** and **C**, representing the row and column dimensions (height and width) that the rectified output image should have.
- Read **four coordinate pairs** (x, y) from the terminal, representing the four corners of the source quadrilateral (the distorted Sudoku in the original image).
- Automatically compute the **four destination coordinate pairs** using the provided dimensions L and C, mapping the corners to the edges of the new image: $(0, 0)$, $(C-1, 0)$, $(0, L-1)$, and $(C-1, L-1)$.
- Load the local image `sudoku.png` and convert it to grayscale.
- Compute the homography matrix and apply the spatial transformation to the image.
- **Output:** Calculate and print the **sum of all pixels** of the resulting image.

📌 **Important**:

- **Input file:** The image `sudoku.png` must be in the same folder as the script. The program must read it directly from disk (e.g., using `mm.read("sudoku.png")` or `cv2.imread`).
- **Point Order:** Ensure that the reading of the 4 source points and the generation of the 4 destination points strictly follow the same corner order: Top-Left (TL), Top-Right (TR), Bottom-Left (BL), and Bottom-Right (BR).
- **Dimensions in OpenCV:** Remember that functions such as `cv2.warpPerspective` expect the output image size in the format `(width, height)`, which is equivalent to `(C, L)`.
- **Interpolation:** To ensure mathematical consistency of the pixel sum with the automatic grader, use the default bilinear interpolation (`flags=cv2.INTER_LINEAR`).
- **Credits:** The image used is **"Sudoku en periódico"** by Héctor Rodríguez, licensed under **CC BY 2.0**.

2.12.11.1 🧠 Problem Context

The homography has 8 degrees of freedom, requiring at least 4 point correspondences to be computed. Unlike affine transformations, it maps any quadrilateral to another quadrilateral, allowing lines that converge to vanishing points to become parallel again:

Operation	Characteristic	Typical Application
Homography	Projection between planes	Document rectification, scanning of plates and QR Codes.
Inverse Mapping	Scanning from destination to source	Avoids "holes" or empty pixels in the final rectified image.
Warping	Spatial resampling	Correction of lens distortion and panorama stitching.

2.12.11.2 📌 Examples

Input	Output	Observation
500 500 100 120 420 95 80 440 450 460	32982820	The first two inputs are the output dimensions (L and C). The following 4 lines are the (x, y) coordinates of the Sudoku corners in the original image + PAD. The output is the total sum of pixels of the rectified image.
200 200 100 120 420 95 80 440 450 460	5277150	Same source points as the previous example, but generating a smaller output image (200×200). The pixel sum decreases proportionally due to the scale.

2.12.11.3 Sudoku image acquisition and conversion to grayscale

Figure 2.22 shows the reading of the original image followed by its conversion to grayscale and resizing to a 500×500 pixel matrix, preparing the data for the next step.

The perspective correction, applied at Figure 2.23 via the homography matrix, eliminates distortions caused by the camera angle and produces a frontal, regular view of the Sudoku grid.

Figure 2.22: Acquisition of a Sudoku image on the left. On the right, conversion to grayscale and resizing. Credit: Héctor Rodríguez de Guardamar, Spain (CC BY 2.0).

```
import cv2
import numpy as np

# --- 1. Loads the saved image (sudoku.png) ---
img = mm.read("sudoku.png")                # BGR, 500×500

# --- 2. Padding to avoid cutting off vertices ---
PAD = 60
```

```python
img_pad = cv2.copyMakeBorder(
    img, PAD, PAD, PAD, PAD,
    cv2.BORDER_CONSTANT, value=[255, 255, 255]
)

# --- 3. Source points (grid corners in the expanded image) ---
pts1 = np.float32([
    [100, 160],    # TL
    [390, 45],    # TR
    [200, 580],   # BL
    [570, 420],   # BR
])
#     W    H

# --- 4. Destination points (front view 500×500) ---
SIZE = 500
pts2 = np.float32([
    [0,    0],
    [SIZE, 0],
    [0,    SIZE],
    [SIZE, SIZE],
])

# --- 5. Homography and rectification ---
img_rect = mm.perspective_transform(img_pad, pts1, pts2, size=(SIZE, SIZE))

# --- 6. Display ---
mm.show(
    [img_pad, img_rect],
    titles=["Original (with padding)", "Rectified front view"],
    cols=2, figsize=(10, 6), axis=True
)
```

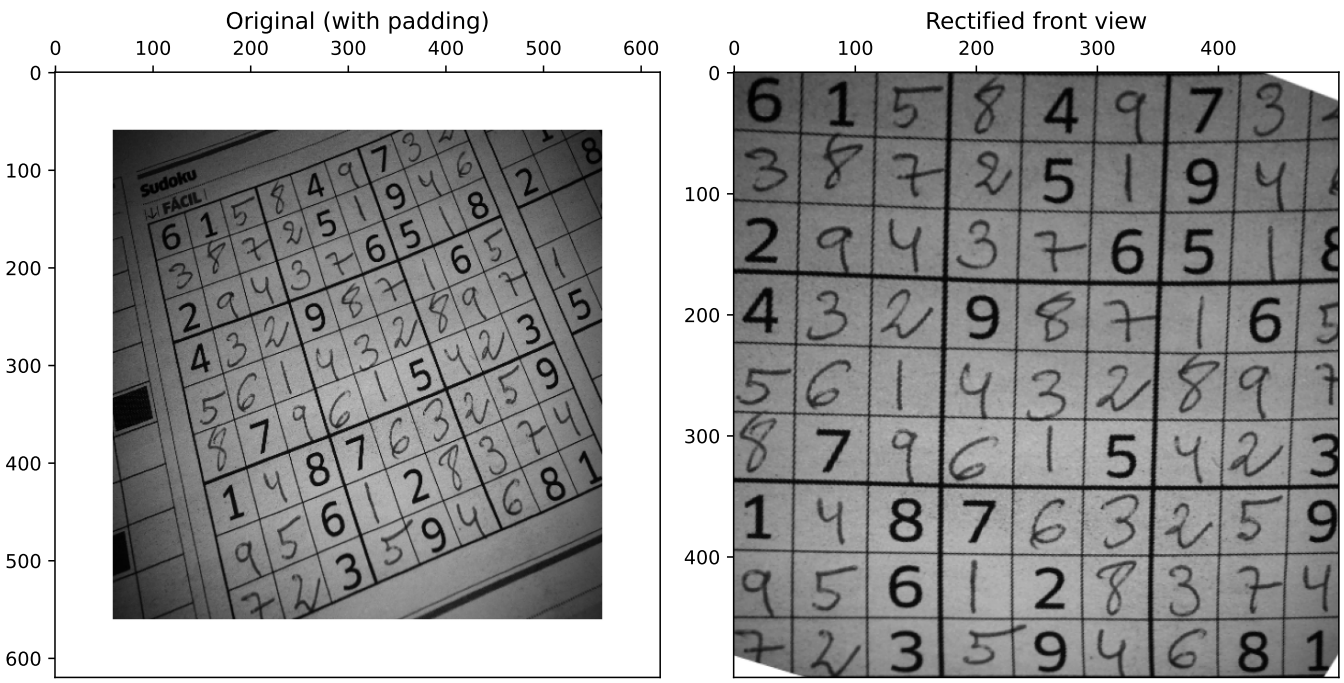
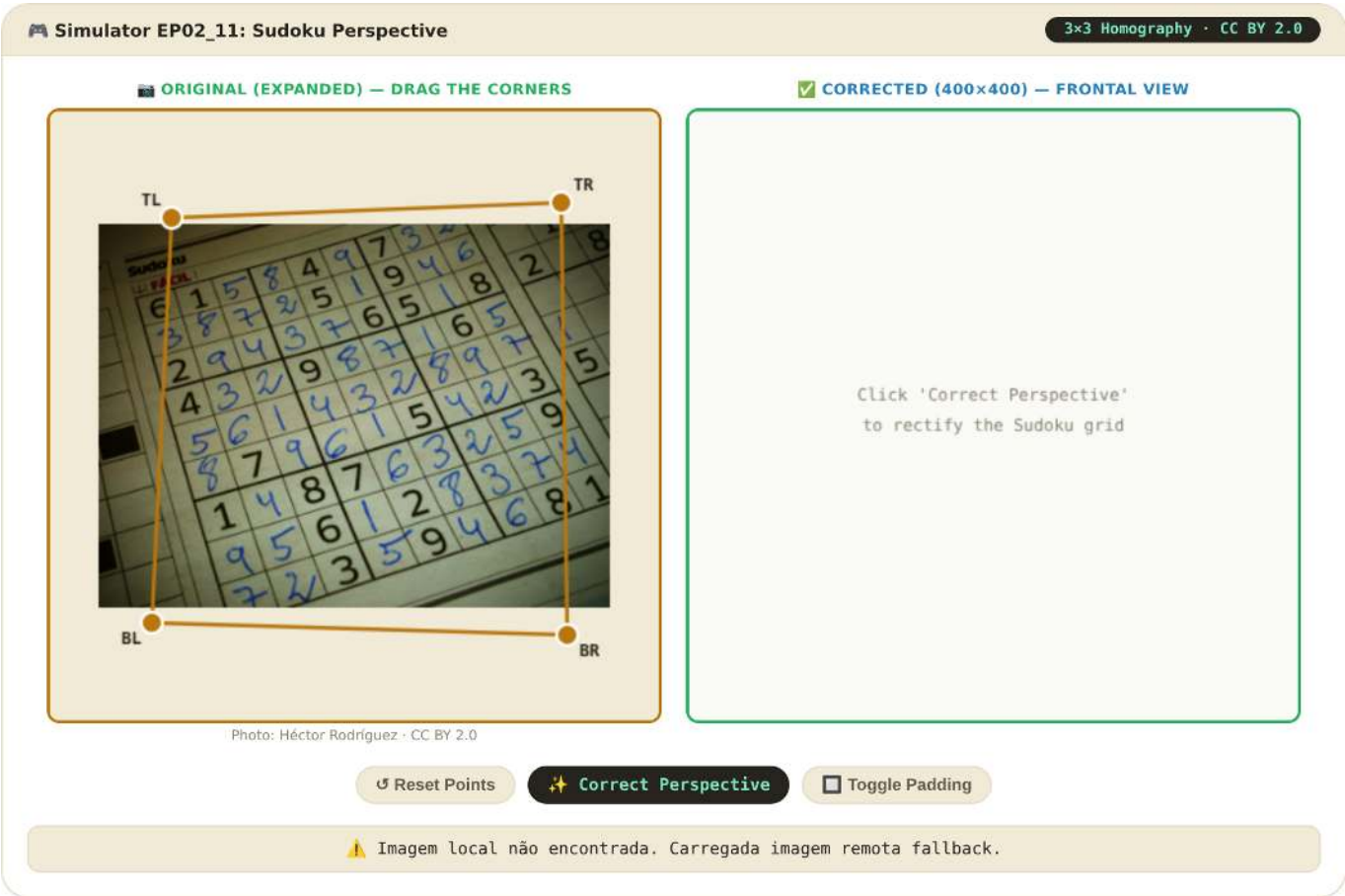


Figure 2.23: Perspective correction: original and rectified front view.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap02/sim-ep0211-sudoku.html

Figure 2.24: EP02_11 Simulator: Sudoku Perspective Correction (3×3 Homography with Bilinear Resampling)

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

```
Overwriting EP02_11.py
```

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

```
<IPython.core.display.HTML object>
```

Chapter 3

Spatial Operations: Intensity, Histogram, and Filtering

Run Colab Open GitHub

This chapter delves into image processing in the spatial domain, starting from the direct manipulation of pixels and histograms for contrast enhancement, to the application of local filters via convolution for smoothing, noise reduction, and edge detection. The goal is to develop the mathematical and computational intuition that underpins much of modern Computer Vision algorithms.

3.1 Objectives

By the end of this chapter, you will be able to:

- **Manipulate intensity and pixels:** Perform saturated arithmetic operations (`mm.addm`, `mm.subm`) and bitwise logic operations (`mm.band`, `mm.bor`, `mm.bnot`) for combining and selecting regions of interest (ROI), and apply *alpha blending* (`mm.blend`) for weighted image fusion;
- **Process histograms:** Interpret the histogram as a tonal diagnostic and apply global equalization via CDF (`mm.equalize`); in the Python track, also adaptive equalization (CLAHE) and histogram specification for tonal profile transfer between images;
- **Understand spatial fundamentals:** Understand neighborhood, border *padding* (`mm.pad`), and the difference between cross-correlation (`mm.conv`) and convolution — including why asymmetric kernels such as Sobel produce distinct results in the two operations;
- **Apply smoothing filtering:** Use the mean filter (`mm.blur`, or `mm.conv` with a uniform kernel) and the Gaussian filter (`mm.gaussian`) for noise reduction, understanding the advantage of radial weighting and Gaussian separability;
- **Apply enhancement filtering:** Use the Laplacian w_4 and w_8 (`mm.laplacian`) for isotropic edge enhancement, the Sobel operator (`mm.sobel`) for gradient magnitude — and, in the Python track, the directional decomposition G_x, G_y, and angle —, and *Unsharp Masking* (`mm.usm`) for high-frequency amplification controlled by the parameter k;
- **Use order filters:** Apply the median filter (`mm.median`) for salt-and-pepper noise removal, understanding why its nonlinear nature and robustness to *outliers* make it superior to linear filters in this scenario;
- **Solve practical problems:** Chain techniques into preprocessing *pipelines* (equalization → Gaussian → Canny; with CLAHE replacing equalization in the Python track) and use the `morph` functions (`mm.conv`, `mm.histImg`, `mm.equalize`, `mm.drawImgKernel`) for didactic analysis and visualization of each step.

3.2 Intensity Level Operations

The most elementary level of image processing acts directly on pixel values, without considering neighborhood. These operations—called **point operations**—are the fastest computationally and form the basis for more complex techniques.

Formally, a point transformation can be described as:

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

where $f(x, y)$ is the input image, $g(x, y)$ is the output, and T is a function applied to each pixel individually.

3.2.1 Preparing the Practical Environment

The following block loads the `morph` library from the repository (the `morph.py` module and, in the C++ track, also the `morph.hpp` used in the `#include` of compiled cells).

```
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup()
from morph import mm
import numpy as np
import cv2
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0
```

As the object of study throughout this chapter, we will use the wildlife images presented in Figure 3.1 and Figure 3.3. From them, we will explore spatial operations on intensity, histograms, and filtering, analyzing their effects on enhancement, smoothing, noise reduction, and edge detection, in order to understand the mathematical and computational fundamentals of DIP.

```
import os

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"

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

img_color = mm.read(caminho)
img_gray  = mm.gray(img_color)

mm.show(img_color)
```

Figure 3.1: *Mandrill* (*Mandrillus sphinx*) photographed in its natural habitat in South Africa. Credit: Carlos Guilherme Rodrigues (CC BY-SA 3.0).

3.2.2 Arithmetic Operations

Arithmetic operations between images are widely used in DIP to combine, compare, or enhance information. **Image subtraction** is especially powerful for detecting differences between two frames — for example, in removing static backgrounds in surveillance cameras:

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

Saturated addition limits the result to the interval $[0, 255]$: values above 255 are clamped to 255, avoiding the silent *overflow* of the `uint8` type (e.g., $200 + 100 = 44$ instead of 300). **Saturated subtraction** applies the same principle on the lower side: negative values are clamped to 0.

> ⚠ Saturation and *overflow*
>
> Arithmetic operations on `uint8` suffer from silent *overflow*: $200 + 100 = 44$ (not 300). `mm.addm` and `mm.subm` perform **automatic saturation**, clamping the result to $[0, 255]$. *Blending* uses fractional weights: `mm.blend` operates internally in floating point and only then rounds and saturates to `uint8`.

Figure 3.2 demonstrates the addition of a constant (brightening) and the subtraction of a constant (darkening with saturation at 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
)
```

Figure 3.2: Saturated arithmetic operations: constant addition (lightening) and constant subtraction (darkening with saturation at 0).

3.2.3 Weighted Mixture (*Alpha Blending*)

The **weighted mixture** (*alpha blending*) combines two images using complementary weights α and $(1 - \alpha)$:

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

When $\alpha = 1$, only image f_1 is obtained; when $\alpha = 0$, only f_2. Intermediate values produce a smooth transition between the two, being widely used in image composition, layer overlaying, watermarks, and visual blending effects.

For the combination to produce a coherent result, it is necessary to align the regions of interest beforehand. In Figure 3.4, the leopard's face is cropped with `mm.crop(img_leop_gray, 250, H-300, 100, W-200)`

and the mandrill's facial region with `mm.crop(img_gray, 100, 400, 380, 530)`, so that the eyes and facial structure are approximately aligned. The leopard crop is then resized (`mm.resize`) to the mandrill's dimensions before the blending.

`mm.blend` performs the operation in floating point — avoiding *overflow* in the calculations with fractional weights — and only then rounds and saturates the result to `uint8`.

```
import os

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

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

img_leop      = mm.read(caminho)
img_leop_gray = mm.gray(img_leop)

mm.show(img_leop)
```

Figure 3.3: Portrait of a leopard (*Panthera pardus*) in its natural habitat. Credit: C. Brück (CC BY-SA 4.0).

```
# aligned crops: leopard face and mandrill facial region
leo      = mm.crop(img_leop_gray, 250, img_leop_gray.shape[0] - 300, 100,
img_leop_gray.shape[1] - 200)
mandrill = mm.crop(img_gray, 100, 400, 380, 530)
leo_r    = mm.resize(leo, (mandrill.shape[1], mandrill.shape[0]), method="bilinear")

mm.show(
    [mm.blend(mandrill, leo_r, alpha=1.0), mm.blend(mandrill, leo_r, alpha=0.8),
     mm.blend(mandrill, leo_r, alpha=0.6), mm.blend(mandrill, leo_r, alpha=0.4),
     mm.blend(mandrill, leo_r, alpha=0.2), mm.blend(mandrill, leo_r, alpha=0.0)],
    titles=[" =1.0", " =0.8", " =0.6", " =0.4", " =0.2", " =0.0"],
    cols=6
```

```
)
```

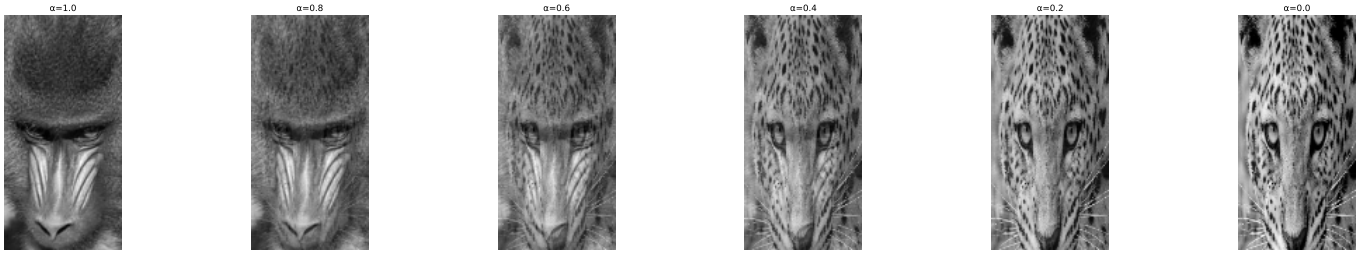


Figure 3.4: *Alpha blending* between aligned crops of mandrill and leopard (Figure 3.3) for different values of . At =1 only the mandrill is seen; at =0, only the leopard; intermediate values merge the gazes of the two images proportionally.

3.2.4 Logical Operations and Bitwise Masks

Bitwise logical operations (AND, OR, and NOT) act directly on the bits of each pixel and are the foundation for creating and applying **masks** (*masks*) — binary images with only 0 (black) and 255 (white) used to isolate **Regions of Interest** (**ROI**).

The behavior of each operation stems from the binary representation of 255 (`11111111`) and 0 (`00000000`):

- **AND** with the mask: where $m = 255$, the original bits are preserved; where $m = 0$, the pixel is zeroed. Result: ROI cropping.

$$g(x, y) = f(x, y) \text{ AND } m(x, y) \tag{3.4}$$

- **OR** with the mask: where $m = 255$, the pixel is forced to white; where $m = 0$, the original value is retained. Result: ROI highlighting.
- **NOT** (without mask): inverts all bits ($g = 255 - f$), producing the photographic negative of the image.

Figure 3.5 illustrates the three operations applied to the mandrill image with a circular mask.

```
h, w = img_gray.shape

# Filled circular mask, centered on the image (mm.circle draws the
# disc; the didactic version, per-pixel radius test, is mm.circle0).
mask_circ = mm.circle(np.zeros((h, w), dtype=np.uint8),
                      (w // 2, h // 2), min(h, w) // 3 - 10, 255, -1)

# Operations via morph
img_not = mm.bnot(img_gray)            # NOT: photographic negative
img_and = mm.band(img_gray, mask_circ) # preserves only the circular ROI
img_or  = mm.bor(img_gray, mask_circ)  # illuminates the mask region

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

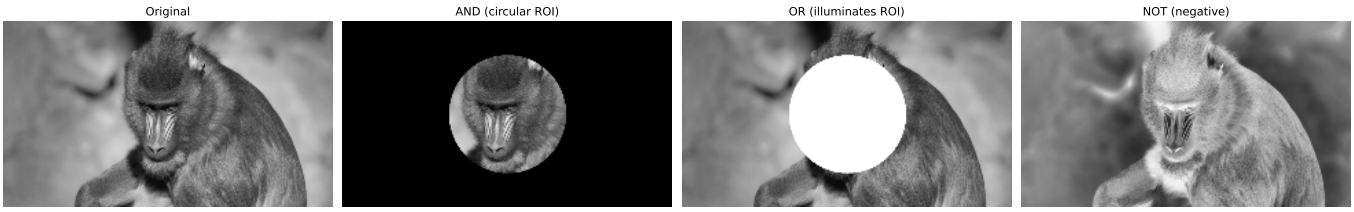


Figure 3.5: Bitwise logical operations with a circular mask: AND (ROI isolation), OR (ROI illumination) and NOT (negative).

3.3 Image Histogram

The **histogram** of a grayscale image is a discrete function that describes the frequency distribution of intensities:

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

where r_k is the k-th intensity level, n_k is the number of pixels with that intensity, and L is the total number of levels (typically 256 for 8 bits). The normalized histogram estimates the probability of each level:

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

where MN is the total number of pixels. As a **global statistic**, the histogram does not carry positional information, but it reveals essential characteristics such as average brightness, contrast, and tonal distribution. In practice, `mm.hist(img)` returns the count vector $h(r_k)$, which serves both for visualization (via `mm.histImg`) and for calculations such as the **cumulative distribution function (CDF)** and equalization.

Histogram Interpretation

- **Narrow on the left:** underexposed image (dark).
- **Narrow on the right:** overexposed image (bright).
- **Concentrated in the center:** low contrast.
- **Spread across the entire range:** high contrast, good utilization of the available tones.

Figure 3.6 presents the histogram of the mandrill image, as well as darkened (`mm.subm`) and brightened (`mm.addm`) versions. The shift of the intensity distribution to the left and to the right is observed, respectively. Note that the range represented on the x-axis does not necessarily correspond to the entire 0 to 255 range.

```
img_dark = mm.subm(img_gray, 80)
img_high = mm.addm(img_gray, 80)

mm.show(
    [img_gray, img_dark, img_high,
     mm.histImg(img_gray), mm.histImg(img_dark), mm.histImg(img_high)],
    titles=["Original", "Darkened (-80)", "Brightened (+80)",
            "Histogram - original", "Histogram - darkened", "Histogram - brightened"],
    cols=3
)
```

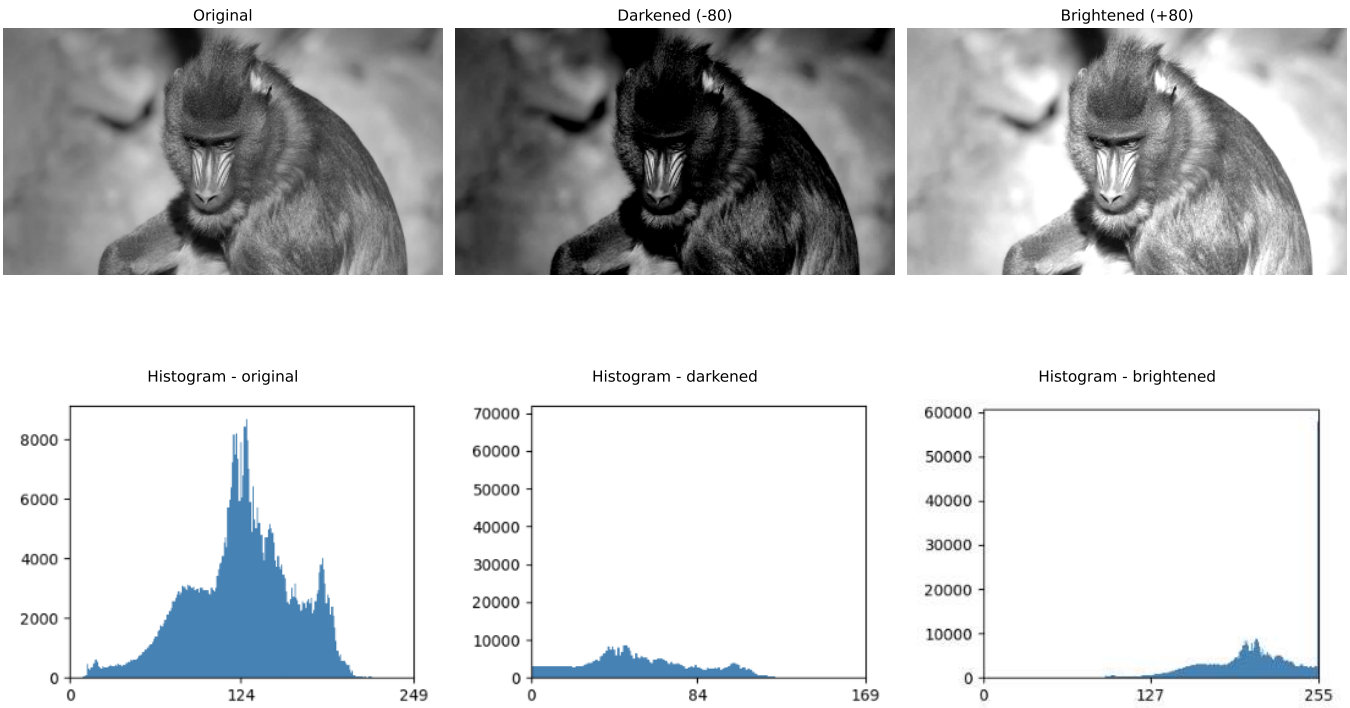


Figure 3.6: Histograms of the original image, a darkened version (−80), and a brightened one (+80). Subtraction/addition saturates at 0 and 255.

3.3.1 Histogram Equalization

Histogram equalization redistributes intensities so that the resulting histogram is as uniform as possible. The mapping is given by the **cumulative distribution function (CDF)**:

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p(r_j) = \frac{L-1}{MN}\sum_{j=0}^{k} n_j \tag{3.7}$$

The transformation is monotonic: frequent levels receive larger intervals in the output domain (greater separation → more contrast), while rare levels are compressed.

The complete algorithm, in five steps, is presented in Table 3.1.

Table 3.1: Histogram equalization algorithm.

Step	Operation	Formula
1	**Histogram**	$h[k] \leftarrow$ number of pixels with intensity k, $k = 0 \ldots L-1$
2	**Probability**	$p[k] \leftarrow h[k]/MN$
3	**CDF**	$\text{cdf}[k] \leftarrow \sum_{j=0}^{k} p[j]$ (cumulative sum)
4	***Look-Up Table* (mapping)**	$\text{lut}[k] \leftarrow \text{round}(\text{cdf}[k] \times (L-1))$
5	**Application**	$g[i,j] \leftarrow \text{lut}[f[i,j]]$ (for every pixel)

Note in Figure 3.7 that equalization **redistributes** the existing tones to more spaced positions in the range $[0, L-1]$, but it does not create new tones — the equalized image still has exactly 3 distinct tones, now at $\{1, 5, 7\}$ instead of $\{2, 3, 4\}$.

```python
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: levels 0..7

# 1. Histogram with size guaranteed up to L
h = mm.hist(img5, 3)  # B=3 → 2³=8 levels, guaranteed size

p   = h / h.sum()  # 2. Probability of each level p[k] = h[k] / MN

cdf = np.cumsum(p) # 3. Normalized CDF or
# 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: mapping to [0, L-1]

img5_eq = lut[img5] # 5. Applies LUT pixel by pixel or, equivalently:
# 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("Original image (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("Equalized image (5×5):")
print(img5_eq)

# Counts distinct tones
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} tones: {sorted(np.unique(img5).tolist())})",
            f"Equalized ({tons_eq} tones: {sorted(np.unique(img5_eq).tolist())})",
            "Histogram - Original",
            "Histogram - Equalized"],
    rows=2, cols=2, figsize=(5, 4), dpi=100
)
```

```
Original image (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

Equalized image (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]]
```

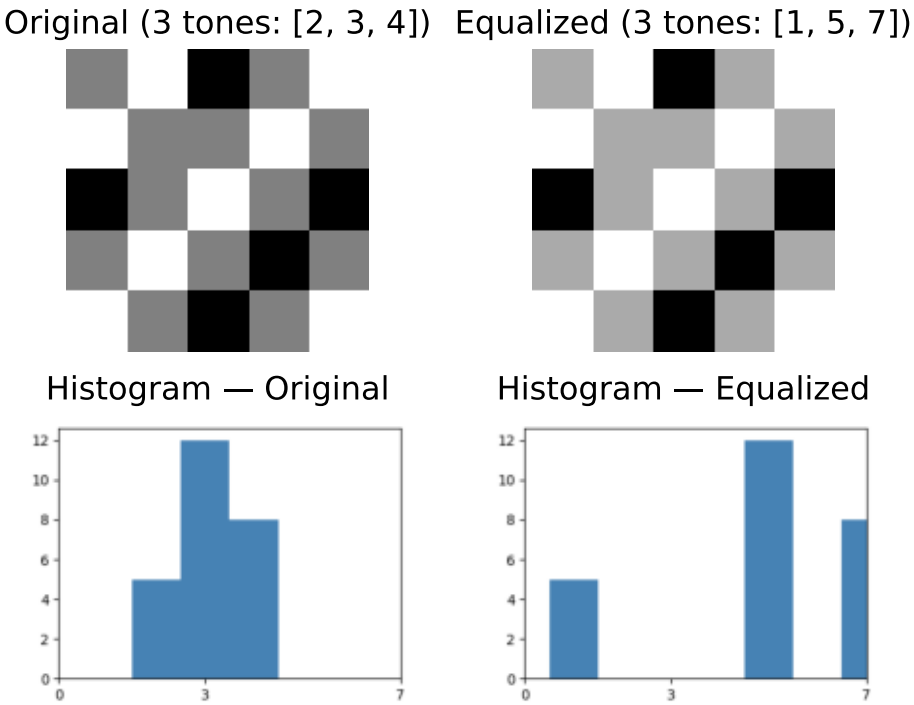


Figure 3.7: Histogram equalization in a 5×5 image with 3 bits (L=8): original image with tones concentrated in {2,3,4}, equalized image with tones redistributed to {1,5,7}, and the respective histograms showing the spreading of frequencies.

Limitation: global equalization can over-enhance noise and produce excessive contrast in homogeneous regions. **CLAHE** (*Contrast Limited Adaptive Histogram Equalization*) reduces this problem by applying equalization in local blocks (*tiles*) and limiting the height of histogram peaks before equalization.

Figure 3.8 compares the original image, the global equalization via `mm.equalize`, and OpenCV's CLAHE, also displaying the resulting histograms. Unlike global equalization, which uses a single transformation based on the CDF of the entire image, CLAHE adapts contrast to each region, being particularly useful in images with non-uniform illumination.

In the example, `clipLimit=2.0` and `tileGridSize=(32,32)` were used. The `clipLimit` parameter defines how much the local histogram peaks can grow before being clipped. In OpenCV, this value is a relative factor: the actual limit is roughly calculated as `clipLimit × (number of pixels in the block / number of gray levels)`. For example, in a block with 4096 pixels and an 8-bit image (256 gray levels), the average frequency per level is $4096/256 = 16$. Thus, `clipLimit=2.0` allows peaks of approximately $2 \times 16 = 32$

occurrences before clipping. The excess occurrences are not discarded: they are redistributed among the remaining gray levels of the histogram, reducing excessive concentration in few levels and avoiding exaggerated amplification of local contrast. Smaller values limit contrast more and reduce noise amplification, while larger values allow more intense enhancement but may introduce artifacts.

- `clipLimit=1.0`: smooth and conservative enhancement;
- `clipLimit=2.0`: good balance between contrast and naturalness;
- `clipLimit=4.0`: greater emphasis on local details;
- `clipLimit=8.0`: aggressive contrast, with possible noise amplification.

Thus, CLAHE usually produces more natural results than global equalization, especially in images with shadows, reflections, or uneven illumination.

```
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)
)
```

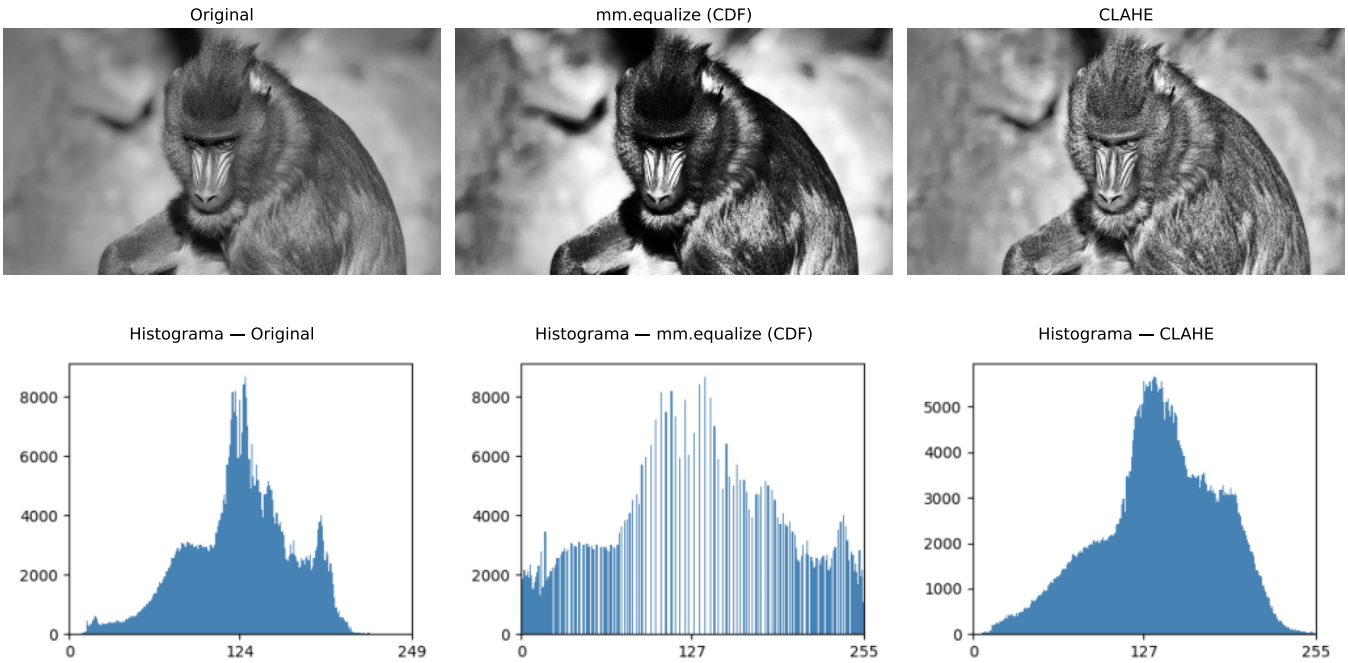


Figure 3.8: Histogram equalization: mm.equalize (global via CDF) vs. CLAHE (adaptive with contrast limitation). The histograms reveal the progressive spread of intensities.

3.3.2 Histogram Specification

While equalization imposes a uniform distribution, **histogram specification** (*histogram matching*) allows the output image's histogram to follow an **arbitrary** distribution — for example, the histogram of another reference image.

The procedure involves three steps:

1. Compute the CDF of the input image: $P_r(r_k)$.
2. Compute the CDF of the reference image: $P_z(z_k)$.
3. For each level r_k, find the level z that minimizes $|P_z(z) - P_r(r_k)|$.

$$T(r_k) = \arg\min_z |P_z(z) - P_r(r_k)| \tag{3.8}$$

In Figure 3.9, we transfer the tonal profile of the leopard (Figure 3.3) to the mandrill image — a direct application of the concept seen in *blending*: instead of blending pixels, here we blend tonal distributions.

```
# maps src to ref tonal profile via CDF
def hist_specify(src, ref):
    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)

mm.show(
    [img_gray, img_leop_gray, img_spec,
     mm.histImg(img_gray), mm.histImg(img_leop_gray), mm.histImg(img_spec)],
    titles=["Mandrill (original)", "Leopard (reference)", "Mandrill → leopard profile",
            "Histogram - Mandrill", "Histogram - Leopard", "Histogram - result"],
    rows=2, cols=3, figsize=(12, 9)
)
```

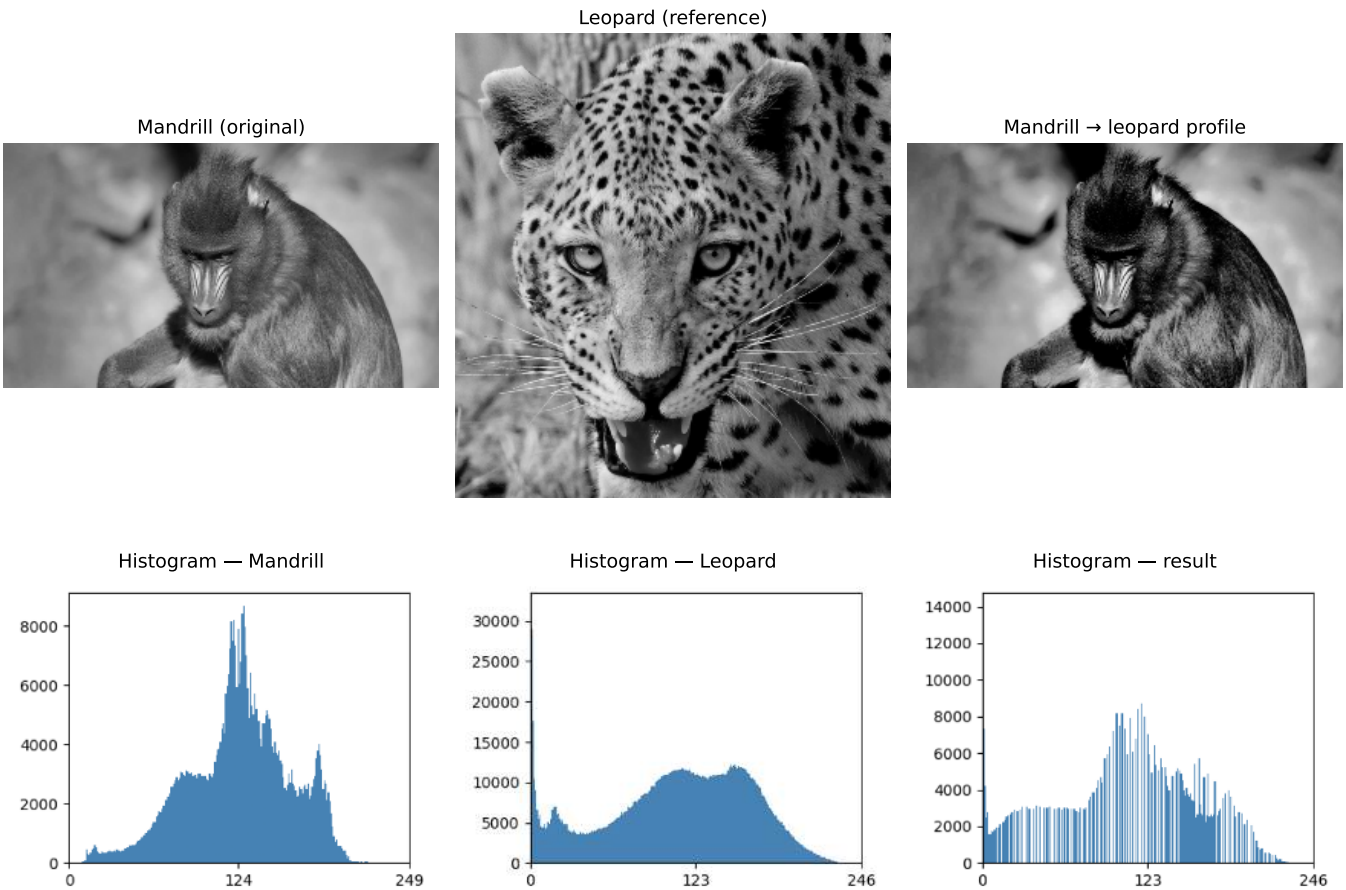


Figure 3.9: Histogram specification: mandrill mapped to the leopard tonal profile. The output CDF approximates the reference CDF.

3.4 Spatial Fundamentals: Neighborhood, Convolution, and *Kernels*

Spatial filtering operations do not act on a single isolated pixel, but on a **neighborhood** around it. For this purpose, a small matrix of coefficients called a **kernel** (or mask) is used, which traverses the entire image by means of a **sliding window**.

The most common windows are 3×3, 5×5, and 7×7. In a 3×3 window, for example, the central pixel is processed together with its eight immediate neighbors. At each window position, the pixel values are combined with the kernel coefficients, producing a new value for the central pixel.

3.4.1 Neighborhood

Consider a 3×3 window centered on the pixel (x, y):

$$\begin{bmatrix} (x-1, y-1) & (x, y-1) & (x+1, y-1) \\ (x-1, y) & (x, y) & (x+1, y) \\ (x-1, y+1) & (x, y+1) & (x+1, y+1) \end{bmatrix} \tag{3.9}$$

In general, a window of size $(2a+1) \times (2b+1)$ encompasses all pixels located up to a positions horizontally and up to b positions vertically relative to the central pixel. Thus, a 3×3 window corresponds to $a = b = 1$, a 5×5 window to $a = b = 2$, and so on.

Mathematically, the neighborhood is defined by

$$\mathcal{V}(x, y) = \{(x+s,\, y+t) : -a \leq s \leq a,\, -b \leq t \leq b\} \tag{3.10}$$

3.4.2 Edge Handling

Pixels near the edges have part of their neighborhood outside the image. To apply filters in these regions, it is necessary to define how the external values will be obtained. The three most common strategies (with the equivalent OpenCV constant in parentheses) are:

- **Zero-padding** (`BORDER_CONSTANT`): fills the external region with zeros.
- **Replication** (`BORDER_REPLICATE`): repeats the value of the border pixel.
- **Reflection** (`BORDER_REFLECT_101`): mirrors the neighboring pixels, without repeating the border pixel.

`mm.conv` uses reflection by default, as it better preserves the continuity of gray levels and reduces artifacts in edge handling.

The following example compares the three strategies with `mm.pad` on a 3×3 matrix. Note how each one fills the external pixels needed to apply a 3×3 filter also at the corners.

```
img = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
], dtype=np.uint8)

bordas = ["constant", "replicate", "reflect101"]

print("Original image:")
print(mm.drawImg(img))

for k in [3, 5]:
    b = k // 2
    print(f"=== Kernel {k}x{k} (padding b={b}) ===")
    for nome in bordas:
```

```
        print(nome)
        print(mm.drawImg(mm.pad(img, b, border=nome)))
```

```
Original image:
1 2 3
4 5 6
7 8 9

=== Kernel 3x3 (padding b=1) ===
constant
0 0 0 0 0
0 1 2 3 0
0 4 5 6 0
0 7 8 9 0
0 0 0 0 0

replicate
1 1 2 3 3
1 1 2 3 3
4 4 5 6 6
7 7 8 9 9
7 7 8 9 9

reflect101
5 4 5 6 5
2 1 2 3 2
5 4 5 6 5
8 7 8 9 8
5 4 5 6 5

=== Kernel 5x5 (padding b=2) ===
constant
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 1 2 3 0 0
0 0 4 5 6 0 0
0 0 7 8 9 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0

replicate
1 1 1 2 3 3 3
1 1 1 2 3 3 3
1 1 1 2 3 3 3
4 4 4 5 6 6 6
7 7 7 8 9 9 9
7 7 7 8 9 9 9
7 7 7 8 9 9 9

reflect101
9 8 7 8 9 8 7
6 5 4 5 6 5 4
3 2 1 2 3 2 1
6 5 4 5 6 5 4
9 8 7 8 9 8 7
6 5 4 5 6 5 4
3 2 1 2 3 2 1
```

Note that the result of a filter can vary significantly depending on the treatment adopted for the image borders.

In the instructional library `morph.py`, the algorithms implemented directly in Python (without OpenCV or *scikit-image*) — the variants with the `0` suffix, such as `mm.conv0` — do not use *padding*. For each pixel, the neighborhood elements are examined individually, and only the neighbors whose coordinates belong to the image domain participate in the calculation. Thus, in regions near the borders, the effective neighborhood may contain fewer pixels than the specified window; in operators based on `mm._viz` (for example, `mm.laplacian`), border pixels with an incomplete window are left unchanged.

3.4.3 Correlation vs. Convolution

There are two mathematically related mechanisms.

Cross-correlation — the *kernel* is applied directly:

$$g(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t)\, f(x + s,\, y + t) \tag{3.11}$$

Two-dimensional convolution — the *kernel* is rotated 180° before application:

$$g(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t)\, f(x - s,\, y - t) \tag{3.12}$$

For symmetric *kernels* (Gaussian, Laplacian, mean), the two operations yield identical results. For asymmetric *kernels* (Sobel, Prewitt), the difference is significant, as shown in the following examples.

3.4.3.1 Correlation (`mm.conv`)

```
# 4x4 image (uint8) and asymmetric 3x3 kernel
img = np.array([[ 1,  2,  3,  4],
                [ 5,  6,  7,  8],
                [ 9, 10, 11, 12],
                [13, 14, 15, 16]], dtype=np.uint8)

w = np.array([[0, 1, 2],
              [0, 0, 0],
              [0, 0, 0]], dtype=np.float32)

corr = mm.conv(img, w, border="constant")  # zero outside the image

print("Original image:");         print(mm.drawImg(img))
print("Kernel:");                  print(mm.drawImg(w))
print("Correlation result:"); print(mm.drawImg(corr))
```

```
Original image:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16

Kernel:
  0   1   2
  0   0   0
  0   0   0

Correlation result:
 0  0  0  0
 5  8 11  4
17 20 23  8
```

```
29 32 35 12
```

`mm.conv` performs correlation, that is, it applies the *kernel* exactly in the provided orientation.

3.4.3.2 Convolution

```
# repeated here so the cell is independent
img = np.array([[ 1,  2,  3,  4],
                [ 5,  6,  7,  8],
                [ 9, 10, 11, 12],
                [13, 14, 15, 16]], dtype=np.uint8)
w   = np.array([[0, 1, 2],
                [0, 0, 0],
                [0, 0, 0]], dtype=np.float32)

# kernel rotated 180° (equivalent to np.rot90(w, 2))
w_conv = np.array([[0, 0, 0],
                   [0, 0, 0],
                   [2, 1, 0]], dtype=np.float32)

conv = mm.conv(img, w_conv, border="constant")

print("Original image:");            print(mm.drawImg(img))
print("Original kernel:");            print(mm.drawImg(w))
print("Kernel rotated 180°:");    print(mm.drawImg(w_conv))
print("Convolution result:");    print(mm.drawImg(conv))
```

```
Original image:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16

Original kernel:
  0   1   2
  0   0   0
  0   0   0

Kernel rotated 180°:
  0   0   0
  0   0   0
  2   1   0

Convolution result:
 5 16 19 22
 9 28 31 34
13 40 43 46
 0  0  0  0
```

The convolution uses the *kernel* rotated by 180°. To reproduce the mathematical definition of convolution, the *kernel* is rotated (here, `[[0,1,2],[0,0,0],[0,0,0]]` → `[[0,0,0],[0,0,0],[2,1,0]]`) before applying `mm.conv`.

3.4.4 The Role of the *Kernel*

The *kernel* coefficients completely determine the effect produced by the filter, as summarized in Table 3.2.

Table 3.2: Typical interpretation of *kernel* coefficients.

Feature	Typical effect
Positive coefficients summing to 1	Smoothing (*low-pass*)
Sum equal to 0, with positive and negative values	Edge detection (*high-pass*)
Dominant central positive coefficient and negative neighbors	Sharpening
Asymmetric coefficients	Directional gradient

Examples:

Smoothing:

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

Edge detection:

$$\begin{bmatrix}-1 & -1 & -1\\ -1 & 8 & -1\\ -1 & -1 & -1\end{bmatrix}$$

Sharpening:

$$\begin{bmatrix}0 & -1 & 0\\ -1 & 5 & -1\\ 0 & -1 & 0\end{bmatrix}$$

Directional gradient (Sobel):

$$\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix}$$

Figure 3.10 demonstrates the step-by-step mechanism: for each position of the window, each coefficient of the *kernel* is multiplied by the corresponding pixel in the neighborhood, and the resulting products are summed. The result is exactly the value defined by Equation 3.11 for that position in the image. Although the images produced by `mm.conv0` and `cv2.filter2D` (or `mm.conv`) are visually very similar, the OpenCV-based implementation is thousands of times faster, as shown below.

⚠ Performance: Python loops vs. vectorized operations

The `mm.conv0` function implements correlation directly in Python through nested loops. Although this approach is suitable for didactic purposes, it executes a large number of operations and becomes slow for larger images.
In contrast, `mm.conv` uses `cv2.filter2D`, implemented in C++ and optimized for matrix operations. In the example presented, the vectorized version was more than **3000 times faster** than the didactic implementation, producing a visually equivalent result.
The numerical differences observed are mainly concentrated at the image borders. In `mm.conv0`, border pixels remain unchanged, while `mm.conv` uses a border reflection strategy (`cv2.BORDER_REFLECT_101`, the default for `cv2.filter2D`).
Therefore, `mm.conv0` should be used to understand the algorithm, while `mm.conv` is the recommended option for practical applications.

```
import time

w_mean = np.ones((3,3), dtype=np.float32) / 9.0
```

```python
img_gray = img_leop_gray
# Numerical demonstration on a 5×5 patch
patch = img_gray[250:255, 250:255].astype(np.float32)
roi   = patch[0:3, 0:3]
print("5×5 patch (intensities):")
print(patch.astype(np.int32))
print(f"\n3×3 averaging kernel:\n{w_mean}")
print(f"\nCorrelation at center pixel [1,1]: \
      {(w_mean * roi).sum():.1f}  (original: {patch[1,1]:.0f})")

# Time comparison on the full image
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"\nTimes on the {img_gray.shape[0]}x{img_gray.shape[1]} image:")
print(f"  mm.conv0 (Python loops): {t_conv0:.3f} s")
print(f"  mm.conv  (cv2.filter2D): {t_conv:.5f} s")
print(f"  Speedup:                {t_conv0/t_conv:.0f}× faster")
print(f"  Maximum difference:    {diff} (borders with different padding)")

mm.show(
    [img_gray, img_conv0, img_conv],
    titles=["Original", f"conv0 ({t_conv0:.2f}s)", f"conv ({t_conv:.4f}s)"],
    cols=3
)
```

```
5×5 patch (intensities):
[[ 91  95 108 116 114]
 [106 107 108 111 114]
 [102 103 107 112 116]
 [ 83  90 111 125 123]
 [ 79  88 110 126 124]]

3×3 averaging kernel:
[[0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]]

Correlation at center pixel [1,1]:       103.0  (original: 107)

Times on the 1324x1242 image:
  mm.conv0 (Python loops): 5.844 s
  mm.conv  (cv2.filter2D): 0.00277 s
  Speedup:                2112× faster
  Maximum difference:    39 (borders with different padding)
```

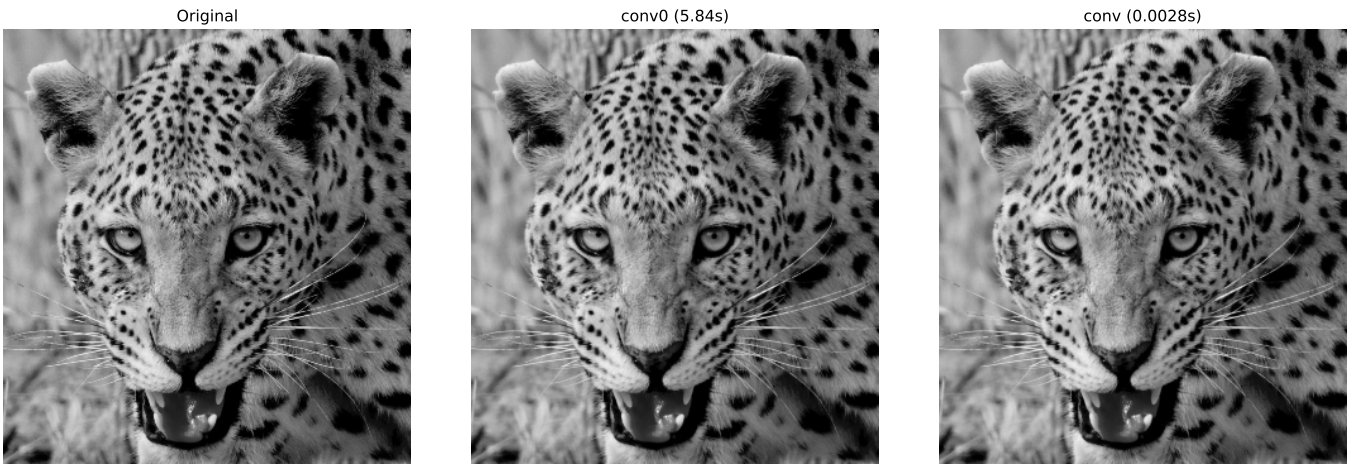


Figure 3.10: Correlation with 3×3 averaging *kernel*: didactic version (mm.conv0) vs. vectorized (mm.conv via cv2.filter2D). The maximum difference of 39 occurs at the borders.

3.4.5 Numerical Example: Step-by-Step Correlation

To make the mechanism of Equation 3.11 concrete, consider the 3×3 averaging *kernel* ($a = b = 1$, all coefficients $= 1/9 \approx 0.111$) applied to the 5×5 *patch* extracted from the leopard image. Figure 3.11 displays the *patch* with a grid and highlights in yellow the 3×3 window centered at pixel $[1, 1]$:

```
patch = img_gray[250:255, 250:255]
B     = np.ones((3, 3), dtype=np.uint8)

print("Patch 5×5 (intensities):")
print(mm.drawImg(patch))

mm.drawImgKernel(patch, B, 1, 1, 40)
```

```
Patch 5×5 (intensities):
 91  95 108 116 114
106 107 108 111 114
102 103 107 112 116
 83  90 111 125 123
 79  88 110 126 124
```

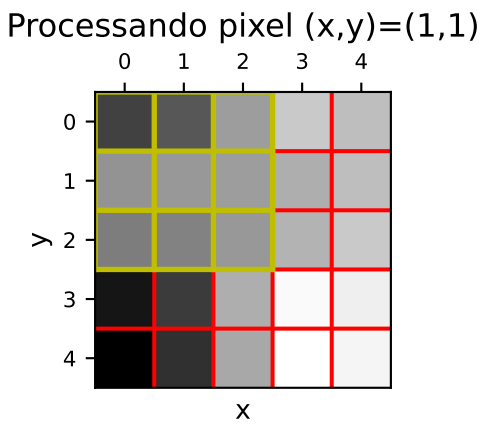


Figure 3.11: *Patch* 5×5 extracted from the leopard image (position [250:255, 250:255]). The yellow window highlights the 3×3 neighborhood centered on pixel [1,1] where the correlation will be computed.

To illustrate the calculation of correlation, consider the pixel at position $[1, 1]$ in the 5×5 *patch* shown in

Figure 3.11. This position was chosen solely for didactic convenience, as it has a complete 3×3 neighborhood around it.

The values in this neighborhood correspond to the upper-left submatrix of the *patch*:

$$\text{neighborhood} = \begin{bmatrix} 91 & 95 & 108 \\ 106 & 107 & 108 \\ 102 & 103 & 107 \end{bmatrix}$$

Applying Equation 3.11 with the mean kernel:

$$g[1,1] = \frac{91 + 95 + 108 + 106 + 107 + 108 + 102 + 103 + 107}{9} = \frac{927}{9} = 103$$

The result (103) is slightly lower than the original value of the central pixel (107), because the mean incorporates neighbors of lower intensity, producing a smoothing effect. In practice, the algorithm starts processing at $[0, 0]$ and repeats this same calculation for each position of the image, shifting the window until it covers the entire domain.

3.5 Smoothing Spatial Filtering

Smoothing filters attenuate abrupt intensity variations, reducing noise and high-frequency details. They are **low-pass** filters — they preserve low-frequency components (large structures) and attenuate high-frequency ones (noise, edges).

3.5.1 Mean Filter (*Box Filter*)

The mean filter uses a uniform kernel of size $n \times n$, where all coefficients equal $1/n^2$:

$$w_{\text{mean}} = \frac{1}{n^2} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}_{n \times n} \tag{3.13}$$

Each output pixel is the arithmetic mean of the n^2 pixels in its neighborhood. Note that the sum of the coefficients is always 1 — the average brightness of the image is preserved. Larger kernels produce more aggressive smoothing but progressively blur the edges.

Figure 3.12 shows the effect of the mean filter with 3×3, 7×7, and 15×15 kernels on a detail of the leopard image. The results were obtained with `mm.blur`, which implements the mean filter using the `cv2.blur` function, equivalent to convolving the image with a uniform kernel whose coefficients are $h(x, y) = 1/N^2$; equivalently, the same result can be obtained with `mm.conv`, computing ($g = f * h$). As the kernel size increases, more pixels contribute to each output value, intensifying smoothing, reducing noise, and making fine details and edges progressively more blurred.

```
# Detail of the eye region
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.blur(img_gray_crop, k) for k in sizes] # or
    #[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)
```

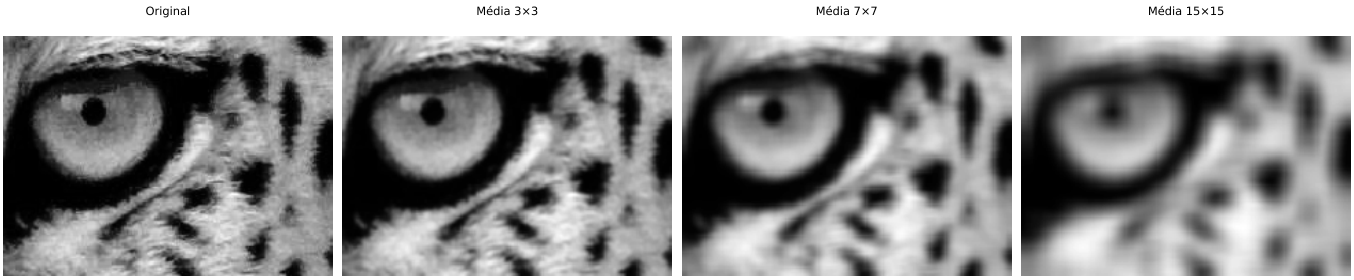


Figure 3.12: Average filter with kernels of increasing size (3×3, 7×7, 15×15). The blurring of the edges increases with the kernel size.

3.5.2 Gaussian Filter

The Gaussian filter weights the pixels in the neighborhood according to a two-dimensional Gaussian function:

$$G(s,t) = \frac{1}{2\pi\sigma^2}\, e^{-\frac{s^2+t^2}{2\sigma^2}} \tag{3.14}$$

where σ is the standard deviation and controls the radius of influence. Pixels closer to the center have greater weight; distant pixels are progressively ignored.

Figure 3.13 presents the Gaussian kernel 5×5 generated for $\sigma = 1$. The kernel was constructed from the outer product of two one-dimensional Gaussian vectors and subsequently normalized so that the sum of its coefficients equals 1. It is observed that the largest weights are concentrated at the center of the matrix, decreasing radially toward the edges. This distribution causes the central pixels to have greater influence on the filtering result, contributing to a more natural smoothing with better edge preservation than the mean filter.

```
# Generates 5×5 Gaussian kernel via cv2
k_gauss = cv2.getGaussianKernel(5, 1)
w_gauss = k_gauss @ k_gauss.T          # @ => matrix multiplication: G(s,t) = G(s)·G(t)
w_gauss = w_gauss / w_gauss.sum()      # ensures sum = 1

print("Gaussian Kernel 5×5 ( =1), normalized:")
for row in w_gauss:
    print("  " + "  ".join(f"{v:.4f}" for v in row))
print(f"\nSum of coefficients: {w_gauss.sum():.6f}")
print(f"Central weight vs. corner: {w_gauss[2,2]:.4f} vs. {w_gauss[0,0]:.4f} "
      f"({w_gauss[2,2]/w_gauss[0,0]:.1f}× larger)")

mm.drawImgPlt((w_gauss * 1000).astype(np.uint8), scale=40)
```

```
Gaussian Kernel 5×5 ( =1), normalized:
  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

Sum of coefficients: 1.000000
Central weight vs. corner: 0.1621 vs. 0.0030 (54.6× larger)
```

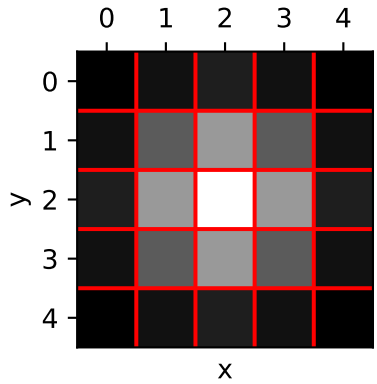


Figure 3.13: *Gaussian* Kernel 5×5 (=1): normalized weights, larger at the center and decreasing radially. The grid facilitates reading each coefficient.

Computational advantage of separability

Consider a square *kernel* of size $n \times n$. If this filter is separable (such as the Gaussian), the 2D convolution can be decomposed into two 1D convolutions: one horizontal and one vertical.
In this case, the cost per pixel decreases from approximately $O(n^2)$ operations (direct 2D convolution) to $O(2n)$ operations (two 1D convolutions). Thus, the complexity is significantly reduced, making processing more efficient

Compared to the averaging filter, the Gaussian:

- **Better preserves edges** — the radial weighting smooths without creating abrupt transitions;
- **Does not introduce ringing** in the frequency domain, since the Gaussian is its own Fourier transform (Chapter 5);
- **Is controlled by** σ — increasing σ is equivalent to increasing the smoothing radius in a continuous and predictable manner.

Figure 3.14 compares the mean and Gaussian filters applied to the leopard image using a 9×9 window. The mean filter was implemented by convolution with a uniform *kernel*, where all 81 pixels in the neighborhood have the same weight (1/81), while the Gaussian filter was obtained with `cv2.GaussianBlur`, using weights defined by a Gaussian distribution. Both reduce noise and smooth the image, but the Gaussian filter better preserves edges and local details, as can be observed in the enlarged region of the eye.

Figure 3.14 compares the mean and Gaussian filters applied to a detail of the leopard image with 9×9 *kernels*. The mean filter was obtained with `mm.blur`, equivalent to convolution with a uniform *kernel* whose coefficients equal 1/81, while the Gaussian filter was obtained with `mm.gaussian`, equivalent to convolution with a *kernel* generated from a Gaussian distribution. Both promote smoothing and noise reduction, but the Gaussian filter assigns greater weight to the central pixels of the neighborhood, better preserving edges and local details, as can be observed in the enlarged region of the eye.

```
img_media9 = mm.blur(img_gray, 9)
img_gauss9 = mm.gaussian(img_gray, 9, 0)

# Detail of the eye region
img_gray_crop  = mm.crop(img_gray,   580, 740, 680, 900)
img_media9_crop = mm.crop(img_media9, 580, 740, 680, 900)
img_gauss9_crop = mm.crop(img_gauss9, 580, 740, 680, 900)

mm.show(
    [img_gray_crop, img_media9_crop, img_gauss9_crop],
    titles=["Detail: Original", "Mean 9×9", "Gaussian 9×9"],
    cols=3, figsize=(12, 8)
```

)

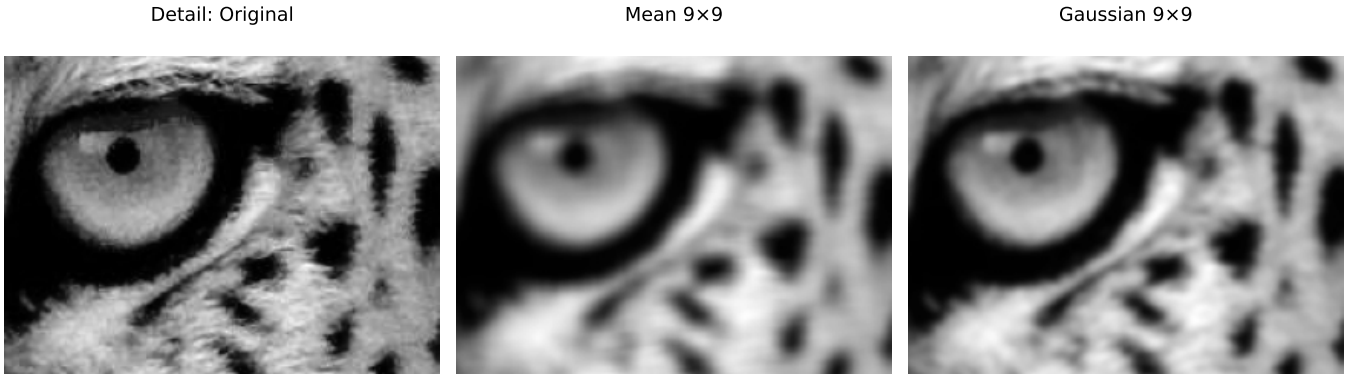


Figure 3.14: Comparison between the mean filter and Gaussian (*kernel* 9×9, =0). The Gaussian preserves edges better, visible in the detail of the face.

3.6 Enhancement Spatial Filtering

Enhancement filters emphasize abrupt intensity transitions, increasing sharpness and edge visibility. They are **high-pass** filters — they amplify high-frequency components (edges, texture) and suppress low-frequency ones (uniform regions).

The intuition is simple: if we subtract from an image its smoothed version (which contains only low frequencies), what remains are the high frequencies — edges and details. Adding this residual back to the original image increases local contrast:

$$g = f + k\,(f - f_{\text{smooth}}), \quad k > 0 \tag{3.15}$$

Figure 3.15 illustrates this process on a synthetic 1D signal with three distinct structures: a wide step, a narrow peak, and a smooth ramp. Panel shows the original signal $f(x)$; panel shows the smoothed version $f_{\text{smooth}}(x)$ obtained by moving average — note how the narrow peak is attenuated. Panel displays the residual $f - f_{\text{smooth}}$, which retains only the abrupt transitions. Finally, panel shows $g(x)$: the previously attenuated peak is restored and amplified relative to the original. Adjust k and the window size to observe the trade-off between sharpness and noise amplification.

Enhancement filters formalize this idea directly in the kernel, without requiring two separate steps.

3.6.1 Laplacian

The Laplacian is an isotropic **second-derivative** operator, meaning it responds equally to variations in all directions, unlike first-derivative operators such as Sobel and Prewitt, which are directional:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{3.16}$$

An important property of the second derivative is that its value is **close to zero in uniform regions** and high at intensity transitions. Thus, by subtracting the Laplacian from the original image, edges and details are enhanced, increasing local contrast:

$$g(x, y) = f(x, y) - \nabla^2 f(x, y) \tag{3.17}$$

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap03/sim-03-filtragem1d.html

Figure 3.15: Simulator: 1D Spatial Enhancement Filtering (Unsharp Masking and High-Boost)

In the discrete form, the second derivative in x is approximated by $f(x+1,y) - 2f(x,y) + f(x-1,y)$, and similarly in y. By summing the two directions, we obtain the *kernel* w_4 (4-neighbor) or w_8 (8-neighbor, including diagonals):

$$w_4 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \qquad w_8 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.18}$$

Zero sum and negative center

Both *kernels* have **coefficient sum equal to zero**: in uniform regions, the output is 0 — the Laplacian does not alter the average brightness, it only detects variations. The **negative center** indicates that the pixel is compared with its neighbors: the more it stands out (upward or downward), the greater the absolute value of the Laplacian at that point.

In the following example, the central pixel $[1,1] = 107$ has neighbors $\{95, 106, 108, 103\}$. Since these values are close to one another, the region is nearly uniform, and the Laplacian returns a low value, producing little enhancement. In edge regions, where there are greater differences between the central pixel and its neighbors, the Laplacian assumes higher values (positive or negative), and the operation in Equation 3.17 intensifies these transitions.

Figure 3.16 illustrates the computation of the Laplacian using the *kernel* w_4 in a 3×3 neighborhood highlighted within a 5×5 patch. The example shows the value obtained by the operator and the corresponding enhanced pixel in the output image, which changes from 107 to 123.

```
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("Original patch:")
print(mm.drawImg(patch))
print(f"Laplac. w4 at [1,1]: ", end="")
print(f"{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"Highlighted pixel [1,1]: {p[1,1]:.0f} - ({lap_11}) = {real_11}")

mm.drawImgKernel(patch, B, x=1, y=1, scale=40.0)
```

```
Original patch:
 91  95 108 116 114
106 107 108 111 114
102 103 107 112 116
 83  90 111 125 123
 79  88 110 126 124

Laplac. w4 at [1,1]: 95 + 106 - 4×107 + 108 + 103 = -16
Highlighted pixel [1,1]: 107 - (-16) = 123
```

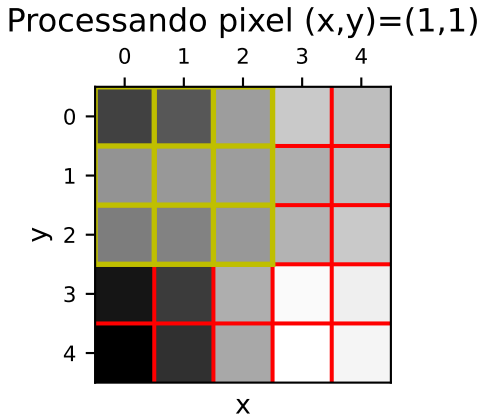


Figure 3.16: *Laplacian kernel* w4 applied to the 5×5 *patch*: the yellow window highlights the 3×3 neighborhood where the second derivative operator is computed.

Figure 3.17 compares the application of the Laplacian *kernels* w_4 and w_8 on a larger crop of the leopard image. For each case, the raw operator response, which highlights edges and intensity transitions, and the image obtained after enhancement by Laplacian subtraction are shown. It is observed that the *kernel* w_8, by also considering diagonal neighbors, produces a stronger response and detects variations in more directions, resulting in a slightly more pronounced enhancement.

```
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)

mm.show(
    [img_gray_crop, mm.laplacian_viz(img_gray_crop, w4), mm.laplacian(img_gray_crop, w4),
     img_gray_crop, mm.laplacian_viz(img_gray_crop, w8), mm.laplacian(img_gray_crop, w8)],
    titles=["Original", "Laplacian w4", "Enhancement w4",
            "Original", "Laplacian w8", "Enhancement w8"],
    cols=3, figsize=(14, 8)
)
```

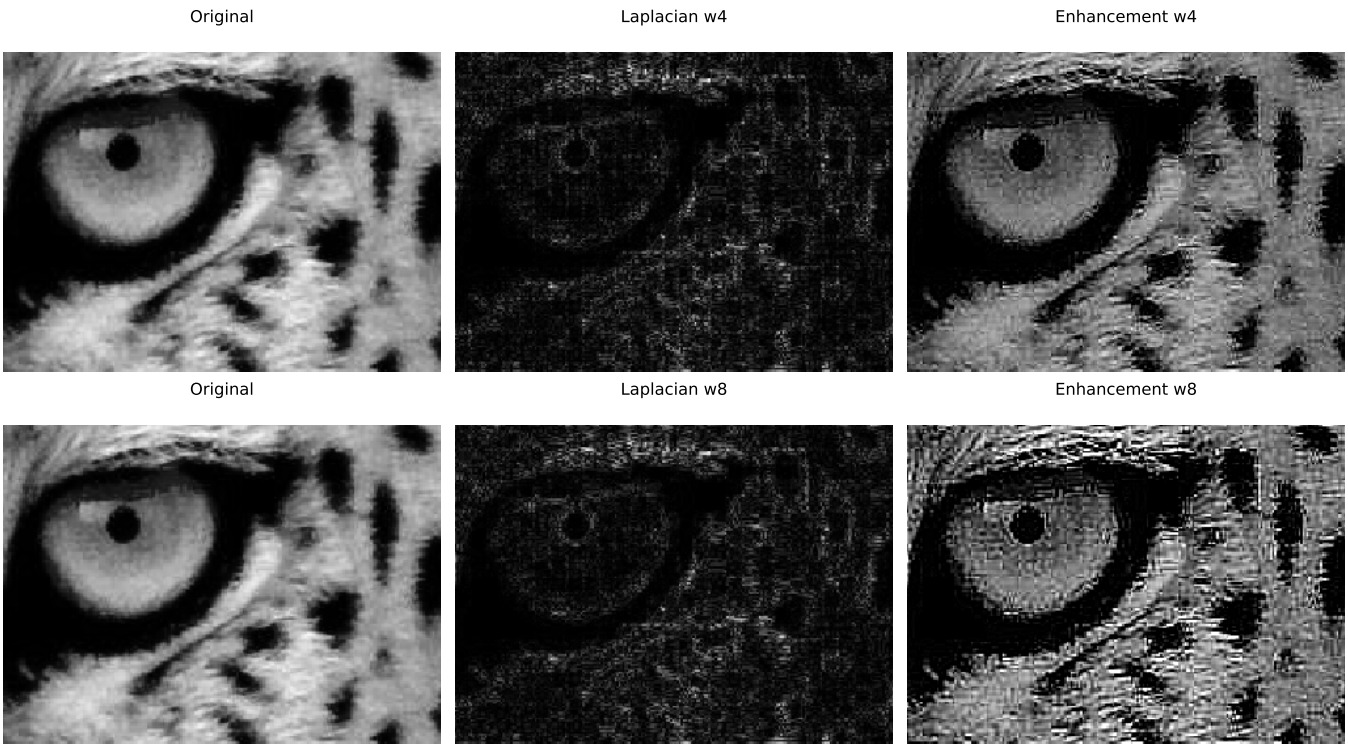


Figure 3.17: Laplacian applied to the leopard image: raw response (edges) with w4 and w8, and images enhanced by subtracting the Laplacian. w8 is more sensitive to diagonals.

3.6.2 Sobel Operator

The Sobel operator estimates the **first-order partial derivatives** in the horizontal and vertical directions. Unlike the Laplacian (second derivative), Sobel is directional and more robust to noise, as each *kernel* combines a derivative with a perpendicular Gaussian smoothing:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * f, \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * f \tag{3.19}$$

G_x detects **vertical** edges (variation in the x direction); G_y detects **horizontal** edges (variation in the y direction). The weights $\{1, 2, 1\}$ in the perpendicular direction correspond to 1D Gaussian smoothing, which reduces sensitivity to noise.

> **ℹ Sobel is correlation, not convolution**
>
> The Sobel *kernels* are **asymmetric** — a 180° rotation alters the result. `cv2.Sobel` implements cross-correlation (like `cv2.filter2D`). To obtain the correct directional derivative, the signs are already defined for correlation: G_x returns positive values where intensity increases from left to right.

The **gradient** magnitude combines the two components, representing edge strength independent of direction:

$$|\nabla f| = \sqrt{G_x^2 + G_y^2} \tag{3.20}$$

And the **direction** of the gradient (perpendicular to the edge) is:

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \tag{3.21}$$

To illustrate numerically, G_x and G_y are computed manually at the central pixel $[1, 1]$ of the 5×5 *patch*:

```
patch = img_gray[250:255, 250:255]
p     = patch.astype(np.float32)

# Gx at pixel [1,1]: right column - left column (weighted)
gx = (p[0,2] + 2*p[1,2] + p[2,2]) - (p[0,0] + 2*p[1,0] + p[2,0])

# Gy at pixel [1,1]: bottom row - top row (weighted)
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.drawImg(patch))
print(f"Gx at [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 at [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}")

# Replaced the | with the mag prefix to not break Quarto's table interpreter
print(f"mag(Grad f)  = sqrt({gx:.1f}^2 + {gy:.1f}^2) = {mag:.1f}")
print(f"Theta        = arctan({gy:.1f}/{gx:.1f}) = {theta:.1f} degrees")
```

```
Patch 5x5:
 91  95 108 116 114
106 107 108 111 114
102 103 107 112 116
 83  90 111 125 123
 79  88 110 126 124

Gx at [1,1]: (108 + 2*108 + 107)       - (91 + 2*106 + 102) = 26.0
Gy at [1,1]: (102 + 2*103 + 107)       - (91 + 2*95 + 108) = 26.0
mag(Grad f)  = sqrt(26.0^2 + 26.0^2) = 36.8
Theta        = arctan(26.0/26.0) = 45.0 degrees
```

The reduced value of $|\nabla f|$ in this *patch* confirms that the region is nearly uniform, since the gradient assumes high values only where there are significant intensity changes. Figure 3.18 applies the Sobel operator to a larger crop of the leopard image. The horizontal (G_x) and vertical (G_y) responses are shown, obtained by convolution with the respective Sobel *kernels*, as well as the magnitude $|\nabla f|$, calculated from the combination of both. While G_x highlights vertical edges and G_y horizontal edges, the magnitude evidences edges in any direction.

```
def norm255(img):
    mn, mx = img.min(), img.max() #+1e-9 to avoid division by zero
    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))

# magnitude = mm.sobel(img_gray_crop) # or, but with clip instead of norm255

mm.show(
    [img_gray_crop, norm255(np.abs(sobelx)), norm255(np.abs(sobely)),
     norm255(magnitude), norm255(direcao % 180)],
    titles=["Original", "Gx (vertical edges)", "Gy (horizontal edges)",
            "Magnitude |f|", "Direction   (mod 180°)"],
```

```
    cols=5
)
```

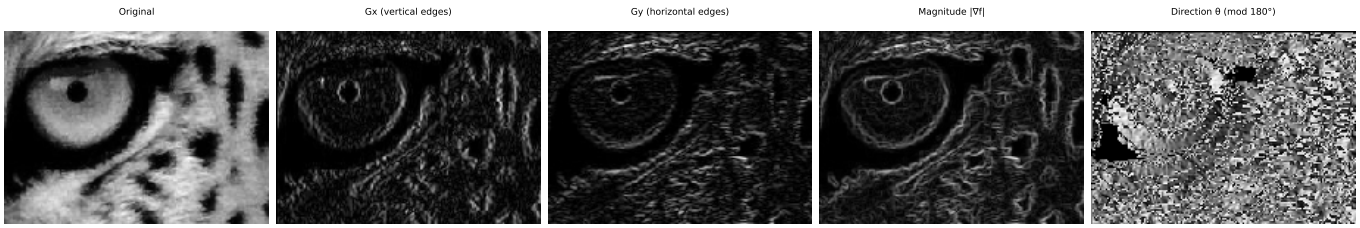


Figure 3.18: Sobel operator on the leopard image: Gx detects vertical edges, Gy detects horizontal edges, and the magnitude | f| combines both, revealing all edges regardless of direction.

3.6.3 Prewitt Operator

The Prewitt operator is structurally identical to Sobel, but replaces the Gaussian weighting $\{1, 2, 1\}$ with uniform weights $\{1, 1, 1\}$:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} * f, \qquad G_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} * f \tag{3.22}$$

The gradient magnitude and direction follow the same equations as Sobel (Equation 3.20 and Equation 3.21). The practical difference is that Prewitt is slightly more sensitive to noise—the uniform perpendicular smoothing weights the central pixel of the line less—but it is computationally simpler. In low-noise images, the results are equivalent.

```
Kx = np.array([[-1,0,1],[-1,0,1],[-1,0,1]], dtype=np.float32)
Ky = np.array([[-1,-1,-1],[0,0,0],[1,1,1]], dtype=np.float32)
f  = img_gray_crop.astype(np.float32)

mm.show(
    [img_gray_crop,
     norm255(np.abs(cv2.filter2D(f, -1, Kx))),
     norm255(np.abs(cv2.filter2D(f, -1, Ky))),
     mm.prewitt(img_gray_crop)],
    titles=["Original", "Gx", "Gy", "Magnitude | f|"],
    cols=4
)
```

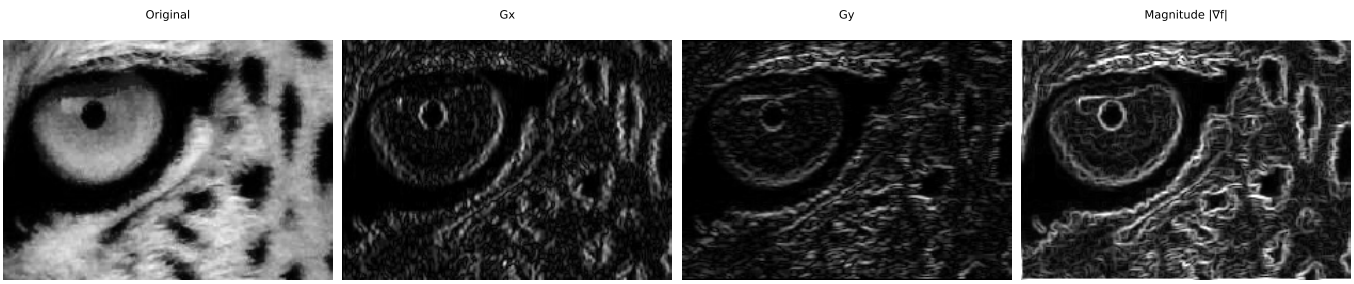


Figure 3.19: Prewitt operator: Gx, Gy and magnitude — comparable to Sobel, but without perpendicular Gaussian weighting.

3.6.4 *Unsharp Masking* (USM)

Unsharp Masking is a classic sharpening technique originating from analog photography, now widely used in image editing software. The central idea is to extract the **high-frequency components** of the image (edges and details) and add them back to the original with a weight k:

Table 3.3: Steps of *Unsharp Masking*.

Step	Operation	Description
1	$\bar{f} = f * G_\sigma$	Smooths with a Gaussian — retains low frequencies
2	$m = f - \bar{f}$	Mask: difference = high frequencies (edges)
3	$g = f + k \cdot m$	Weighted sum of the mask added to the original

Substituting step 2 into step 3 yields the compact expression:

$$g = f + k\,(f - f * G_\sigma) = (1 + k)\,f - k\,(f * G_\sigma) \tag{3.23}$$

The parameter k controls the intensity of the enhancement:

- $k = 0$: no enhancement ($g = f$);
- $k = 1$: classic USM — doubles the contribution of high frequencies;
- $k > 1$: *High Boost Filtering* — amplification beyond double, useful for highly blurred images.

Noise Amplification

USM does not distinguish edges from noise — both are high-frequency components. For high k, the noise present in the image is amplified along with the edges. Therefore, it is advisable to apply slight smoothing before USM on noisy images, or to use a small σ in the Gaussian.

To illustrate the steps of USM, Figure 3.20 applies the method to a 30×30 *patch* of the leopard image, using $\sigma = 1$ and $k = 1$. Initially, the image is smoothed by a Gaussian filter. Then, the high-frequency mask is obtained by the difference between the original and the smoothed image. Finally, this mask is added to the original image, enhancing edges and details. The figure presents the three steps of the process and the final enhancement result.

```
patch  = img_gray_crop[35:65, 45:75]
k = 1.0
sigma = 1.0

p = patch.astype(np.float32)

# Step 1: Gaussian smoothing
ksize   = int(6 * sigma + 1) | 1  # binary operator ensures odd size
p_suave = mm.gaussian(patch, ksize, sigma).astype(np.float32)

# Step 2: high-frequency mask
mascara = p - p_suave

# Step 3: enhancement
p_usm = np.clip(p + k * mascara, 0, 255).astype(np.uint8)

# p_usm = mm.usm(patch, k)  # or, directly
```

```
print(f"Central pixel [1,1]: original={int(p[1,1])} smoothed={p_suave[1,1]:.1f}"
      f" mask={mascara[1,1]:.1f} enhanced={p_usm[1,1]}")

mm.show(
    [patch,
     p_suave.astype(np.uint8),
     np.clip(mascara + 128, 0, 255).astype(np.uint8),  # visible negatives
     p_usm],
    titles=["Original patch",
            f"Smoothed ( ={sigma})",
            "Mask (high freq., +128)",
            f"USM enhanced (k={k})"],
    cols=4, figsize=(12, 4)
)
```

```
Central pixel [1,1]: original=16 smoothed=20.0 mask=-4.0 enhanced=12
```

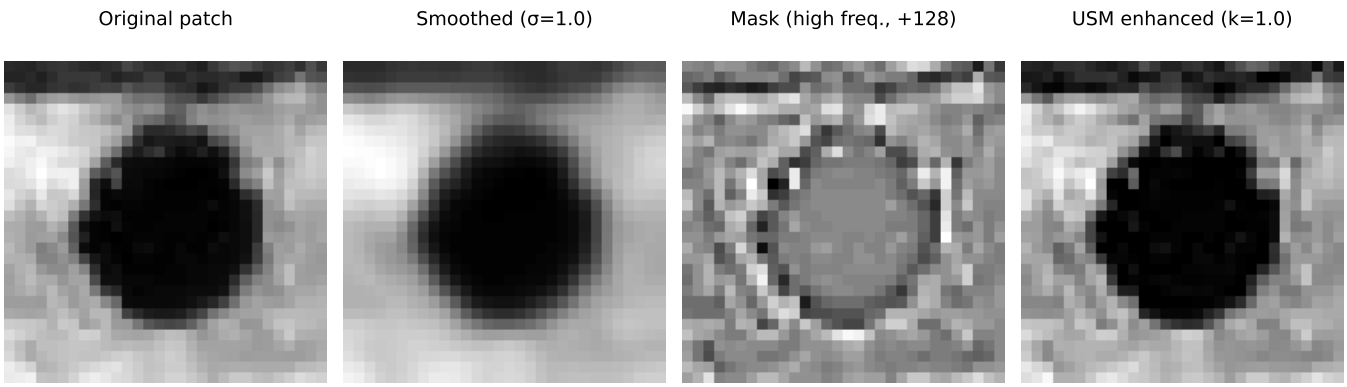


Figure 3.20: Sharpening by *Unsharp Masking* on the leopard image: the smoothed image is subtracted from the original to generate the high-frequency mask, which is then reintroduced to enhance edges and details.

Figure 3.21 applies the USM method to a larger crop of the leopard image using $\sigma = 1$ and different values of the gain factor k. In all cases, the high-frequency mask is obtained by the difference between the original image and its version smoothed by a Gaussian filter. The parameter k controls the intensity of the enhancement: smaller values produce a subtle increase in sharpness, while larger values progressively reinforce edges and details. It is observed that, for high values of k, halos appear around edges and the noise present in the image becomes amplified.

```
mm.show(
    [img_gray_crop,
     mm.usm(img_gray_crop, 0.5), mm.usm(img_gray_crop, 1.0),
     mm.usm(img_gray_crop, 3.0), mm.usm(img_gray_crop, 5.0),
     mm.usm(img_gray_crop, 8.0)],
    titles=["Original", "USM k=0.5", "USM k=1.0", "USM k=3.0", "USM k=5.0", "USM k=8.0"],
    cols=3
)
```

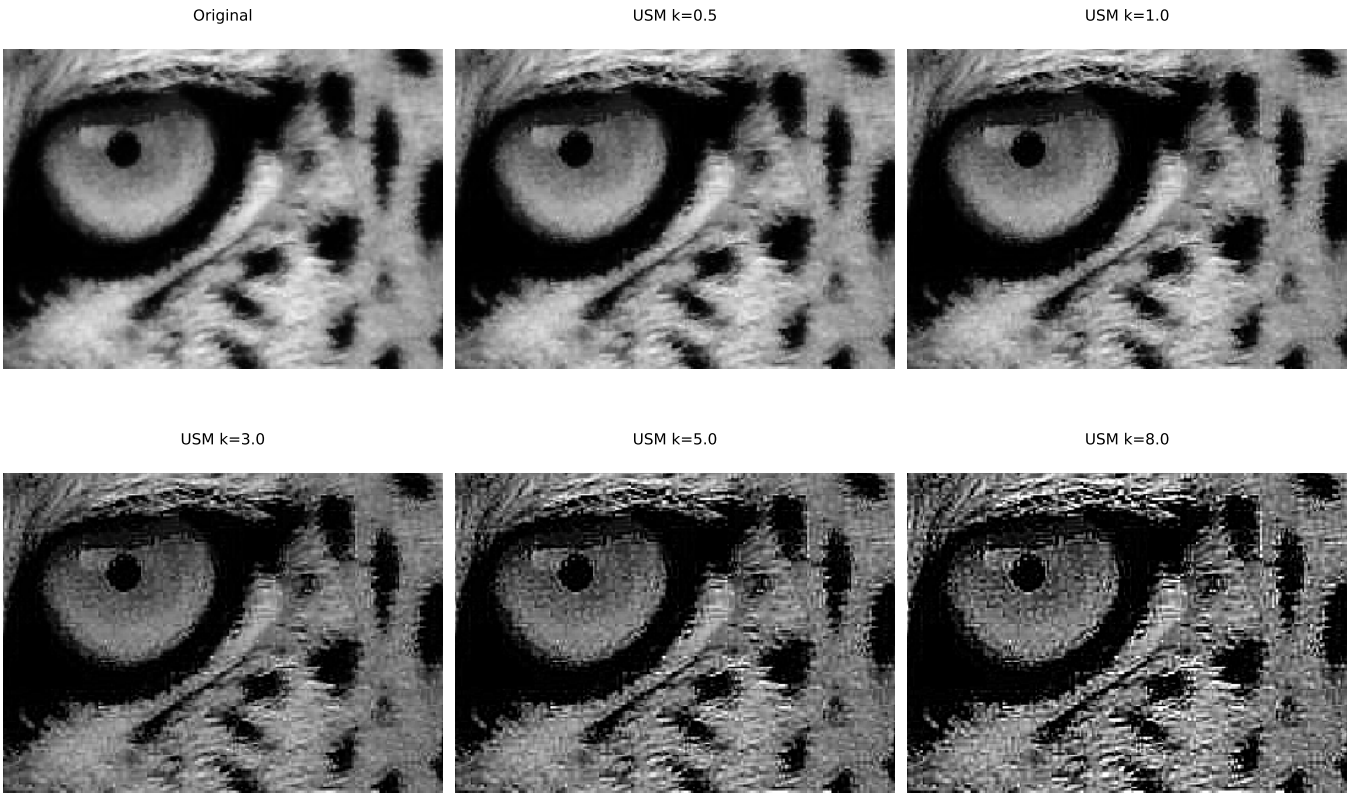


Figure 3.21: *Unsharp Masking* on the leopard image with =1 and k from 0.5 to 8.0. For k>2 halos appear at the edges and the background noise becomes visible.

3.6.5 Canny Detector

Canny combines four steps in sequence — Gaussian smoothing, Sobel gradient, non-maximum suppression, and dual-threshold hysteresis — to produce **thin, binary, and connected** edges. Unlike Sobel and Prewitt, the result is not a continuous gradient map, but a mask where each pixel is either an edge or not.

The central parameter is the threshold pair (T_{low}, T_{high}). Pixels with gradient above T_{high} are certain edges; below T_{low}, they are discarded. The ambiguous pixels — between the two thresholds — are decided by **hysteresis**: they become edges if they are connected to a certain edge, and are discarded otherwise. This avoids both the loss of weak segments of real edges and the inclusion of isolated noise. A common heuristic is $T_{high} = 3 \times T_{low}$.

```
mm.show(
    [img_gray_crop,
     mm.canny(img_gray_crop, 30,  90),
     mm.canny(img_gray_crop, 60,  180),
     mm.canny(img_gray_crop, 120, 240)],
    titles=["Original", "Canny (30/90)", "Canny (60/180)", "Canny (120/240)"],
    cols=4
)
```

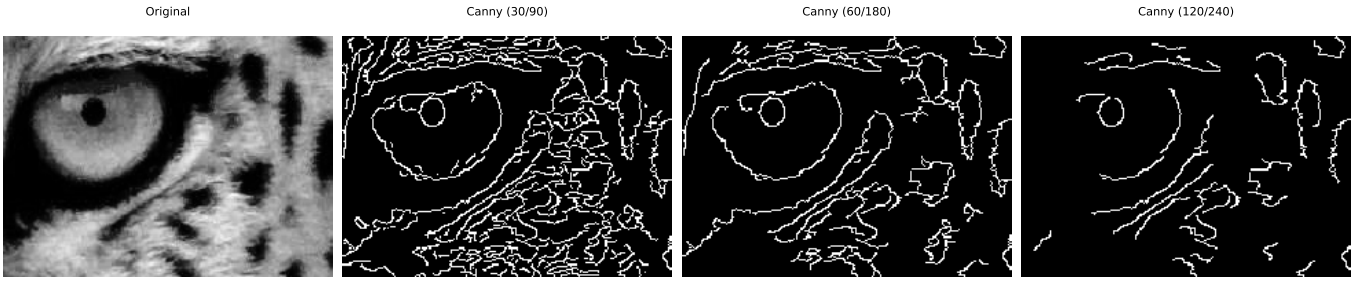


Figure 3.22: Canny detector with different threshold pairs: low thresholds capture more edges (including noise); high thresholds retain only the strongest edges.

Threshold Selection

A common heuristic is $T_{high} = 3 \times T_{low}$. Typical values depend on the image's gradient range — `cv2.Canny` accepts absolute values in $[0, 255]$. For images with variable contrast, computing the thresholds from the percentiles of the Sobel magnitude is more robust than using fixed values.

3.7 Order Filters: Median Filter

Order-statistic filters replace the central pixel with the value of a **percentile** of the neighborhood intensity distribution—unlike linear filters, which compute weighted combinations. The most important one is the **median filter**.

3.7.1 Impulsive Noise: Salt and Pepper

Salt-and-pepper noise replaces random pixels with extreme values: 0 (pepper, black) or 255 (salt, white). It is common in image transmission with bit errors and in cameras with defective sensors.

To understand why linear filters fail, consider a 3×3 neighborhood where a single pixel has been corrupted to 255:

$$\text{neighborhood} = \begin{bmatrix} 102 & 98 & 105 \\ 100 & \mathbf{255} & 97 \\ 103 & 99 & 101 \end{bmatrix}$$

Table 3.4: Mean vs. median with one corrupted pixel. The median ignores the outlier; the mean is shifted by ~40 levels.

Method	Calculation	Result
Mean	(102+98+...+255+...+101)/9	140
Median	{97,98,99,100,**101**,102,103,105,255}	101

Why do mean filters fail with impulsive noise?

The mean is sensitive to ***outliers***—a single pixel with a value of 255 in a neighborhood with values 100 raises the output to 140, spreading the noise throughout the image. The median, being a **robust estimator**, selects the central value of the sorted distribution, naturally discarding the extremes without any special adjustment.

The following example illustrates the behavior of the mean and the median in the presence of a pixel corrupted by impulsive noise. It is observed that the mean is strongly influenced by the extreme value (255), producing

an estimate far from the predominant values of the neighborhood. Meanwhile, the median remains close to the original value of the region, evidencing its greater robustness to *outliers* and justifying its use in salt-and-pepper noise removal.

```
# Numerical example: mean vs. median with corrupted pixel
vizinhanca = np.array([102, 98, 105, 100, 255, 97, 103, 99, 101])
print(f"Neighborhood: {sorted([int(x) for x in vizinhanca])}")
print(f"Mean:      {vizinhanca.mean():.1f}  (shifted by the outlier 255)")
print(f"Median:    {int(np.median(vizinhanca))}    (ignores the outlier)")
print(f"Original value:  ~100")
```

```
Neighborhood: [97, 98, 99, 100, 101, 102, 103, 105, 255]
Mean:      117.8  (shifted by the outlier 255)
Median:    101    (ignores the outlier)
Original value:  ~100
```

Figure 3.23 shows the effect of salt-and-pepper noise at different densities. The noise was generated by randomly replacing a fraction of pixels with minimum values (0, pepper) and maximum values (255, salt). As the density increases from 2% to 10%, the number of corrupted pixels grows, making the visual degradation more evident and hindering the perception of image details.

```
def add_salt_pepper(img, prob=0.05, seed=42):
    """Adds salt and pepper noise with total probability prob."""
    noisy = img.copy()
    rnd   = np.random.default_rng(seed).random(img.shape)
    noisy[rnd < prob / 2]      = 0    # pepper
    noisy[rnd > 1 - prob / 2] = 255  # salt
    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)
```

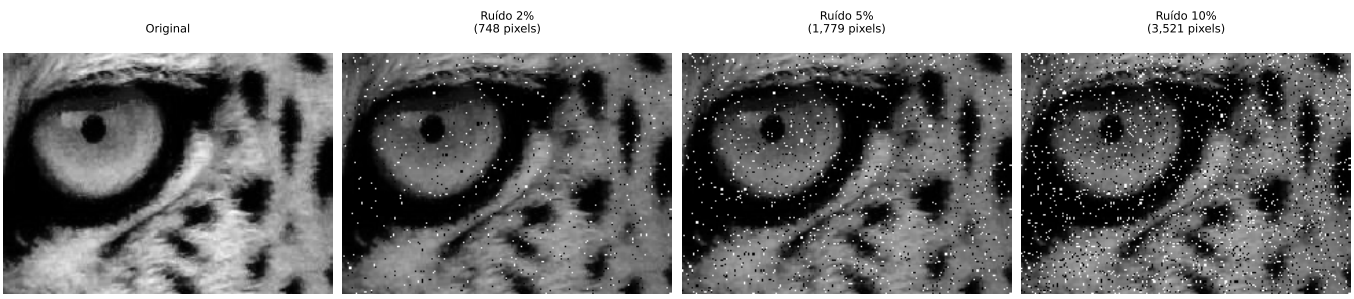


Figure 3.23: Salt and pepper noise with increasing densities (2%, 5%, 10%). The parameter prob indicates the total fraction of corrupted pixels, half salt (255) and half pepper (0).

3.7.2 Median Filter

The median filter replaces each pixel with the **median value** of the pixels in its $n \times n$ neighborhood:

$$g(x, y) = \text{med}_{(s,t)\in\mathcal{V}_n}\{f(x + s, y + t)\} \tag{3.24}$$

The median value is the one occupying the central position when the n^2 values of the neighborhood are sorted. For a 3×3 window ($n^2 = 9$ pixels), the median is the 5th value in the sorted sequence.

To illustrate, consider the same 5×5 *patch* with a pixel artificially corrupted at $[1, 1]$:

```
patch    = img_gray[250:255, 250:255].copy()
corrompido = patch.copy()
corrompido[1, 1] = 255   # inject a salt pixel at the center

# 3×3 neighborhood centered at [1,1]
viz_orig = sorted(patch[0:3, 0:3].ravel().tolist())
viz_corr = sorted(corrompido[0:3, 0:3].ravel().tolist())

print("Original patch:")
print(mm.drawImg(patch))
print("\nPatch with corrupted pixel [1,1]=255:")
print(mm.drawImg(corrompido))
print(f"\nOriginal 3×3 neighborhood (sorted):   {viz_orig}")
print(f"Original median:                       {viz_orig[4]}")
print(f"\nCorrupted 3×3 neighborhood (sorted): {viz_corr}")
print(f"Corrupted median:                     {viz_corr[4]}  ← ignores the 255")
print(f"Corrupted mean:                    {np.mean(viz_corr):.1f}  ← distorted by the 255")
```

```
Original patch:
 91  95 108 116 114
106 107 108 111 114
102 103 107 112 116
 83  90 111 125 123
 79  88 110 126 124


Patch with corrupted pixel [1,1]=255:
 91  95 108 116 114
106 255 108 111 114
102 103 107 112 116
 83  90 111 125 123
 79  88 110 126 124


Original 3×3 neighborhood (sorted):   [91, 95, 102, 103, 106, 107, 107, 108, 108]
Original median:                       106

Corrupted 3×3 neighborhood (sorted): [91, 95, 102, 103, 106, 107, 108, 108, 255]
Corrupted median:                     106  ← ignores the 255
Corrupted mean:                    119.4  ← distorted by the 255
```

The example confirms: even with the pixel corrupted to 255, the median returns the correct central value — the *outlier* occupies the last position in the ordering and is naturally discarded.

Because it is based on ordering rather than summation, the median has three fundamental properties that distinguish it from linear filters:

- **Robust** to impulsive noise — outliers go to the ends of the ordered sequence and do not affect the central value;
- **Edge-preserving** — abrupt intensity transitions are maintained, since the median selects a value that already exists in the neighborhood, without creating new intermediate levels;
- **Nonlinear** — it cannot be expressed as a convolution, therefore `mm.conv` does not apply; use `cv2.medianBlur`.

Figure 3.24 compares different salt-and-pepper noise removal techniques applied to an image with 10% corrupted pixels. The Gaussian, Mean, Median, Bilateral, and Morphological filters (next chapter) were

evaluated, allowing observation of the trade-off between noise removal and detail preservation. In general, the mean and Gaussian filters reduce noise but tend to blur edges, while the median filter performs better for impulsive noise. The bilateral filter preserves edges better, and the morphological filter removes a good portion of corrupted pixels without excessively degrading the image structure.

```
# 10% noise to highlight differences between filters
noisy_5, _ = add_salt_pepper(img_gray_crop, prob=0.1)

#   Filters
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)

# morphological filters in the next chapter, but already shown here for comparison
# B         = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
# f_morf    = cv2.morphologyEx(           # presented in the next chapter
#                 cv2.morphologyEx(noisy_5, cv2.MORPH_OPEN,  B),
#                 cv2.MORPH_CLOSE, B)
B = mm.secross()  # 3×3 box structuring element
f_morf     = mm.asf(noisy_5, 'OC', B)  # equivalent via morph

#   MSE
def mse(a, b):
    return float(np.mean((a.astype(np.float32) - b.astype(np.float32))**2))

print(f"{'Filter':<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)
)
```

```
Filter                     MSE
--------------------------------
Gaussiano 5×5            260.84
Média 5×5                872.57
Mediana 3×3               31.26
Mediana 5×5               65.86
Bilateral               1001.22
Morf. open+close          65.16
```

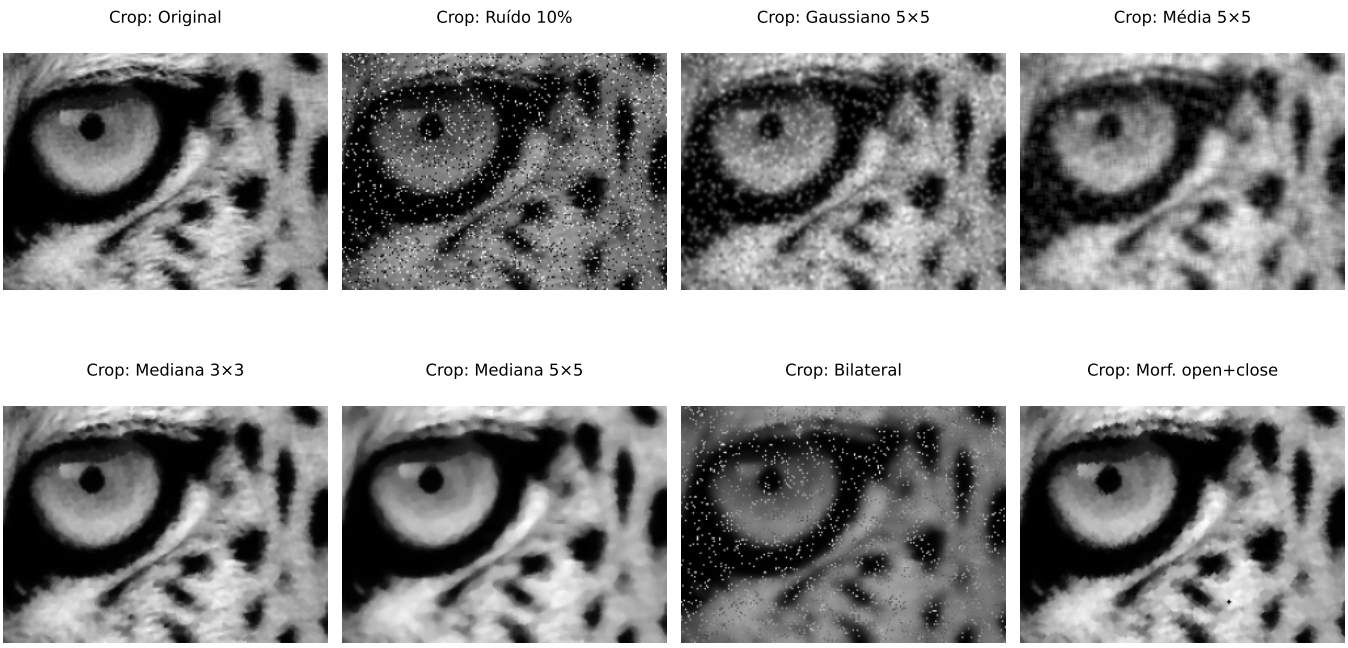


Figure 3.24: Comparison of filters for removing salt-and-pepper noise (10%): Gaussian, Mean, Median, Bilateral, and Morphological. Top row: full images; bottom row: crop of the region of interest.

3.8 Practical Application: Preprocessing for Segmentation

In practice, the techniques in this chapter are rarely used in isolation. A typical **preprocessing pipeline** combines multiple steps in sequence, tailored to the image type and application. Figure 3.25 illustrates a complete pipeline:

1. **Histogram equalization (CLAHE):** normalizes contrast regardless of lighting conditions;
2. **Gaussian filter:** smooths acquisition noise without destroying edges;
3. **Edge detection (Sobel/Canny):** extracts relevant structures for segmentation.

> **ℹ Order matters**
>
> The order of operations affects the final result. In general: **(1) intensity normalization → (2) noise reduction → (3) enhancement/segmentation**. Reversing the order can amplify noise or cause edges to be lost before detection.

```
# Step 1: CLAHE
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
img_clahe = clahe.apply(img_gray_crop)

# Step 2: Gaussian
img_gauss = cv2.GaussianBlur(img_clahe, (5, 5), 0)

# Step 3: Canny
edges = cv2.Canny(img_gauss, 50, 150)

# Comparison: pipeline vs. direct Canny
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. Gaussian", "3. Canny (pipeline)", "Canny (direct)"],
    cols=5
```

```
)
```

Figure 3.25: *Pipeline* de pré-processamento: CLAHE → Gaussiano → Canny. Cada etapa prepara a imagem para a seguinte, resultando em bordas limpas e bem definidas.

3.9 Summary

This chapter presented the main spatial domain processing techniques, from direct pixel manipulation to neighborhood filtering:

- **Point operations:** saturated arithmetic (`mm.addm`, `mm.subm`) and bitwise logic (`mm.band`, `mm.bor`, `mm.bnot`) for ROI cropping and image combination; *alpha blending* (`mm.blend`) for weighted fusion with weight $\alpha \in [0, 1]$.
- **Histogram:** discrete function of intensity distribution; visualized with `mm.histImg` and computed with `mm.hist`; the basis for tonal diagnosis and for equalization and specification techniques.
- **Equalization:** automatic redistribution of intensities via the CDF (`mm.equalize`), with the adaptive CLAHE variant for local contrast control.
- **Histogram specification:** transfer of the tonal profile from a reference image via inverse CDF mapping — a generalization of equalization to arbitrary distributions.
- **Correlation and convolution:** sliding window mechanism implemented in `mm.conv` (`cv2.filter2D`); distinguished by the 180° rotation of the *kernel* — relevant only for asymmetric kernels.
- **Smoothing filters:** mean (uniform *kernel*, blurs edges proportionally to size) and Gaussian (radial weighting, separable, no *ringing*, preserves edges better).
- **Sharpening filters:** Laplacian (w_4/w_8, isotropic second derivative), Sobel (first-order directional gradient, with magnitude $|\nabla f|$ and direction θ) and *Unsharp Masking* (high-frequency amplification with parameter k).
- **Median filter:** nonlinear, robust to outliers, edge-preserving — superior to linear filters for salt-and-pepper noise.
- **Practical pipeline:** CLAHE → Gaussian → Canny chaining as a preprocessing strategy; `mm.drawImgKernel` for didactic visualization of the sliding window.

Chapter 4 will address **mathematical morphology** (erosion, dilation, opening, and closing), exploring in depth the `mm.ero` and `mm.dil` functions from the `morph.py` library. Next, Chapter 5 will present **frequency domain processing**, focusing on the Fourier Transform and spectral filtering techniques.

3.10 🤖 Using Gemini Notebook as a Complementary Tutor

In this edition, we encourage the use of **Gemini Notebook** as a complementary learning tool. This AI tool uses exclusively the documents provided by the author as its knowledge base, ensuring responses consistent with the book's content—including the functions of the `morph.py` library and the experiments conducted in this chapter.

For each chapter, we have prepared a specific project on the platform containing the chapter's PDF, the *notebooks*, and auxiliary materials. We suggest exploring in particular:

- **Study Guide:** a structured summary of the concepts, ideal for review before exams;
- **Chat:** ask questions about equalization, convolution, filters, and pipelines directly with the tutor;

- **Frequently Asked Questions:** typical questions about the difference between mean and median, USM, Laplacian vs. Sobel.

🎓 Study with the Intelligent Tutor

To interact with the content of this chapter, access the following *link*. The environment contains teaching materials in different formats, generated from the chapter's **PDF**. On the platform, explore especially the **Study Guide** and **Chat** options to deepen your understanding.
🚀 ACCESS GEMINI NOTEBOOK: CHAPTER 03

🌐 **Language and Programming Language**

The project for this chapter in Gemini Notebook was built using only the text in **Portuguese** and the code examples in **Python**. If you are studying from the English or French edition, or following the C++ track, the tutor's answers may not correspond exactly to the version you are reading.

⚠️ **Notice on AI-Generated Content**

AI is a powerful ally in your studies, but the generated content may contain **errors or inaccuracies**. Always consult **books, scientific articles, and other reliable academic sources** to validate the information. Whenever possible, run the practical examples provided in this chapter to verify the results.

3.11 Exercise List

1. **(10%)** Explain the difference between **convolution** and **cross-correlation**. For which types of *kernels* are the results identical? Give an example of an asymmetric *kernel* (such as Sobel G_x) and show numerically that the results differ by applying it to the 5×5 patch from the chapter in both ways.

2. **(15%)** Consider a 5×5 image with intensities concentrated between levels 3 and 5 (low contrast, 3 bits). Manually apply the equalization algorithm from Table 3.1, filling in all the columns of the table (k, $h[k]$, $p[k]$, cdf$[k]$, lut$[k]$). Verify the result with `mm.equalize`.

3. **(15%)** Using `mm.conv`, apply the mean filter with kernels of size 3×3, 9×9, and 21×21 to the mandrill image. For each version, compute the **PSNR** (*Peak Signal-to-Noise Ratio*) relative to the original:

$$\text{PSNR} = 10\log_{10}\left(\frac{255^2}{\text{MSE}}\right), \quad \text{MSE} = \frac{1}{MN}\sum_{i,j}(f-g)^2$$

   Plot the PSNR as a function of kernel size and explain what the progressive decline indicates about the relationship between smoothing and information loss.

4. **(15%)** Using `add_salt_pepper` with a density of 5%, apply and compare: (a) `mm.conv` with a 3×3 mean filter, (b) `cv2.GaussianBlur` with $\sigma = 1$, (c) `cv2.medianBlur` with a 3×3 window, and (d) `cv2.medianBlur` with a 5×5 window. Display the images with `mm.show` in a 2×4 grid (row 1: images, row 2: histograms via `mm.histImg`). Explain why the median outperforms linear filters using the argument from Table 3.4.

5. **(15%)** Implement `mm.conv0` using only vectorized NumPy operations — without Python loops and without `cv2.filter2D` — using the *stride tricks* operator (`np.lib.stride_tricks.sliding_window_view`). Compare the result and execution time with `mm.conv0` (loops) and `mm.conv` (cv2) for 3×3 and 15×15 kernels on the mandrill image.

6. **(15%)** Apply *Unsharp Masking* with $\sigma = 1$ and $k \in \{0.5, 1.0, 2.0, 4.0\}$ using the `usm` function from the chapter. For each value of k: (a) compute the absolute difference $|g - f|$, (b) display the images and the differences with `mm.show`, and (c) plot the histogram of the differences with `mm.histImg`. Identify from which k onward the artifacts (halos and noise amplification) become visually unacceptable.

7. **(15%)** Choose a publicly available X-ray or tomography image (e.g., via `mm.read` from a URL) and design a preprocessing pipeline with at least 4 sequential steps, justifying each choice based on the concepts from the chapter. Display with `mm.show` in a grid: the original image, each intermediate stage, and the final result along with their histograms (`mm.histImg`).

Chapter References

The theoretical foundation of this chapter is based on the following works:

- Gonzalez (2018) for the concepts of intensity operations, histogram, convolution, and spatial filtering.
- Szeliski (2022) for computer vision and practical applications of filtering.
- Bradski (2008) for the practical implementation with OpenCV and `morph.py`.

Run Colab Open GitHub

3.12 Exercises Proposed (EPs)

3.13 Homework: Implement and Test the Following Functions

3.14 💻 Practical Part with Programming Exercises

🎯 Objective of this Notebook

The notebook allows you to develop, validate, organize, and test solutions for **Programming Exercises (EPs)** in interactive environments, such as Colab, using the same test cases as Moodle, and only copy them there when recording the official grade.

Download

Download `morph.py` and `testsuite.py` by running the cell below:

```
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup(testsuite=True)
from morph import mm
from testsuite import TestSuite
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Running the Tests

To evaluate the tests, run `TestSuite("EP03_01.extension").run()` in a new cell, replacing the extension with the one corresponding to the language used (`.py`, `.java`, `.c`, `.cpp`, `.js`, or `.r`). The system downloads the test cases from GitHub, runs the program, and calculates the grade automatically.

To test Python code directly, without saving a file, use `run_code(code)` passing the code as a *string* in a variable `code`:

```
code = """
from morph import mm
# ... your code here ...
"""
TestSuite("EP03_01").run_code(code)
```

3.14.1 EP03_01 ➕ Saturated Addition of a Constant

In video surveillance systems, cameras in environments with variable lighting produce underexposed images. Adjusting brightness by **saturated addition of a constant** is the simplest operation for immediate correction, being applied in real time on embedded camera *chips* and in preprocessing *pipelines* of mobile robots.

See Figure 3.26 for a simulation of this EP.

3.14.1.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Constant:** Read the integer k (value to be added).
3. **Data:** Read the integer values of the original matrix row by row.
4. **Mapping:** For each pixel p, compute the new value using the equation:

$$p' = \text{clip}(p + k)$$

5. **Output:** Display the resulting matrix with dimensions $L \times C$.

3.14.1.2 📌 Computational Constraints

- **Saturation (*Clipping*):** Values must be confined to the interval $[0, 255]$:

$$\text{clip}(x) = \max(0, \min(255, x))$$

- **Type:** The final result must be an integer (no decimal places).
- **k can be negative:** negative values darken the image; positive values lighten it.

3.14.1.3 🧠 Theoretical Background

Parameter	Type	Visual Impact
$k > 0$	Integer	Lightens the image; pixels near 255 saturate to white
$k < 0$	Integer	Darkens the image; pixels near 0 saturate to black
$k = 0$	Integer	Image unchanged

3.14.1.4 📦 Input and Output Specification (VPL)

Input:

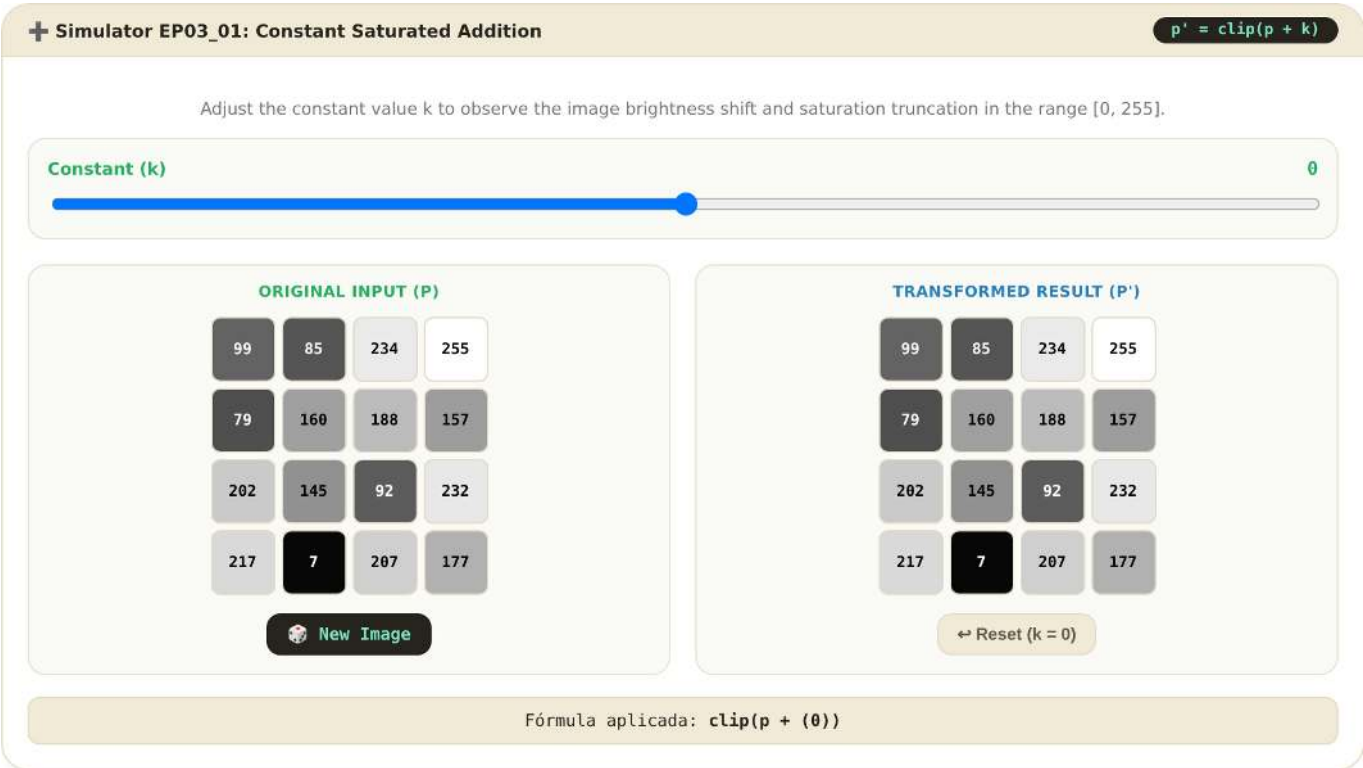
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer k.
- Following lines: Integer elements of the original matrix.

Output:

- Transformed matrix with L rows and C columns, integer values separated by spaces.

3.14.1.5 📌 Examples

Input	Output	Observation
2 3 50 0 100 200 210 240 255	50 150 250 255 255 255	Saturation at 255 for high pixels
1 4 -30 0 20 200 255	0 0 170 225	Saturation at 0 for low pixels



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap03/sim-ep0301-adicao.html

Figure 3.26: Simulador EP03_01: Saturated Addition of Constant (p' = clip(p + k))

```
%%writefile EP03_01.py
# Python code
```

```
Overwriting EP03_01.py
```

```
TestSuite("EP03_01.py").run()
```

```
<IPython.core.display.HTML object>
```

3.14.2 EP03_02 🔀 Alpha Blending of Two Images

In nuclear medicine, images from different modalities (computed tomography and magnetic resonance imaging) are fused to aid in diagnosis. **Weighted blending** (*alpha blending*) is the fundamental operation of this process, allowing the radiologist to interactively control the weight of each modality in the displayed image.

See Figure 3.27 for a simulation of this EP.

3.14.2.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Parameter:** Read the real value $\alpha \in [0, 1]$.
3. **Data:** Read the integer values of matrix f_1 (image 1) followed by those of matrix f_2 (image 2).
4. **Mapping:** For each position (i, j), compute:

$$g(i, j) = \text{clip}\left(\text{round}\left(\alpha \cdot f_1(i, j) + (1 - \alpha) \cdot f_2(i, j)\right)\right)$$

5. **Output:** Display the resulting $L \times C$ matrix.

3.14.2.2 📌 Computational Constraints

- **Rounding:** Apply `round` before converting to integer.
- **Saturation:** Constrain to the interval $[0, 255]$ with $\text{clip}(x) = \max(0, \min(255, x))$.
- **Float operation:** Perform the operation in floating point before rounding.

3.14.2.3 🧠 Theoretical Background

Value of α	Result
$\alpha = 1.0$	Only f_1
$\alpha = 0.5$	Arithmetic mean of f_1 and f_2
$\alpha = 0.0$	Only f_2

3.14.2.4 📦 Input and Output Specification (VPL)

Input:

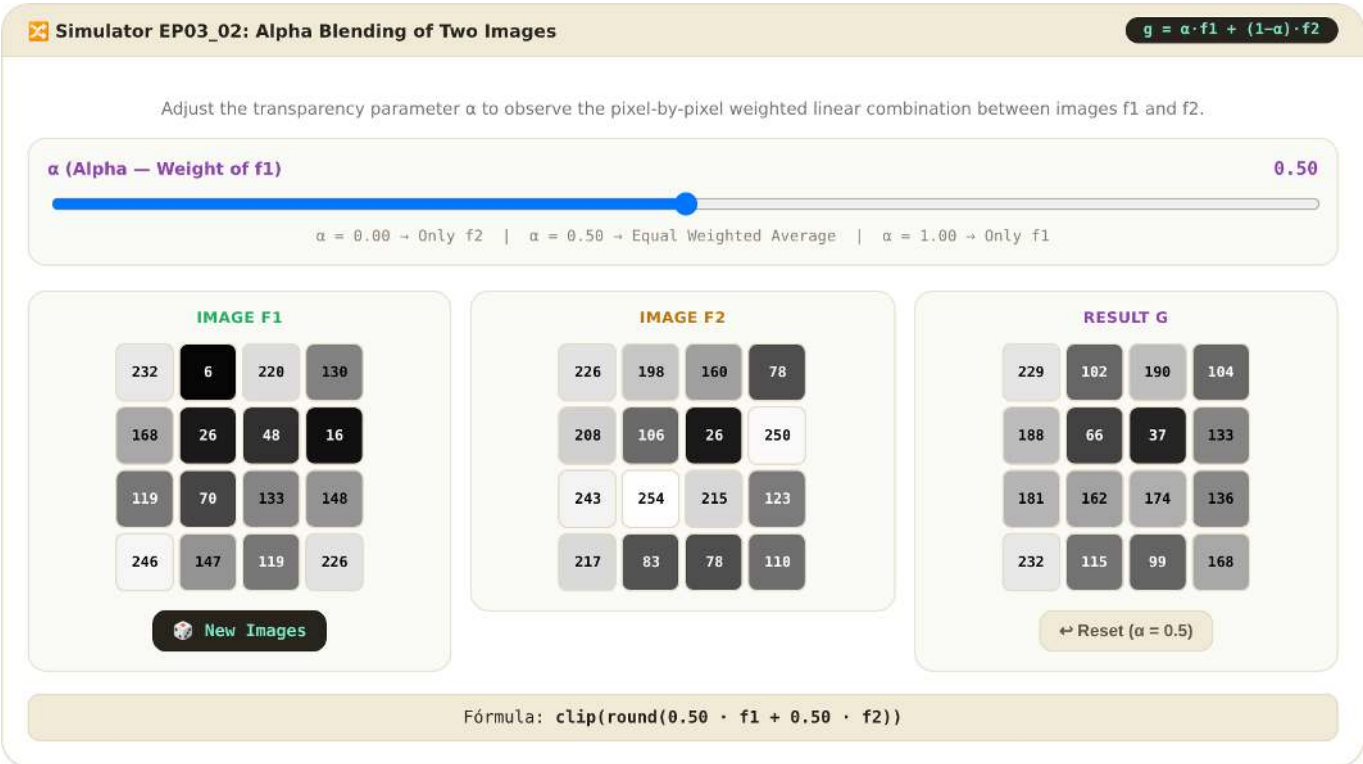
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Real α.
- Following lines: Elements of f_1 (L lines with C values each).
- Following lines: Elements of f_2 (L lines with C values each).

Output:

- Resulting $L \times C$ matrix.

3.14.2.5 📌 Examples

Input	Output	Observation
1 3 0.5 0 100 200 100 200 50	50 150 125	Mean between the two images
1 3 1.0 10 20 30 90 80 70	10 20 30	Only f_1 (alpha=1)



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap03/sim-ep0302-blending.html

Figure 3.27: Simulator EP03_02: Alpha Blending of Two Images (g = · f1 + (1−) · f2)

```
%%writefile EP03_02.py
# Python code
```

```
Overwriting EP03_02.py
```

```
TestSuite("EP03_02.py").run()
```

```
<IPython.core.display.HTML object>
```

3.14.3 EP03_03 🎭 Image Inversion (Photographic Negative)

In radiology, X-ray images are traditionally viewed as negatives: bones appear in black on a white background. The **photographic negative** operation is routinely applied in PACS (*Picture Archiving and Communication Systems*) to facilitate the detection of fractures and bone densities.

See Figure 3.28 for a simulation of this EP.

3.14.3.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Data:** Read the integer values of the original matrix.
3. **Mapping:** For each pixel p, compute the negative:

$$p' = 255 - p$$

4. **Output:** Display the resulting matrix $L \times C$.

3.14.3.2 📌 Computational Constraints

- **No clipping required:** The result of $255 - p$ with $p \in [0, 255]$ is always $\in [0, 255]$.

- **Integer type:** The output must consist of integer values.
- **Logical equivalence:** The operation is identical to the bitwise `NOT` (`mm.bnot`) on 8-bit images.

3.14.3.3 🧠 Theoretical Foundation

Original Pixel p	Negative Pixel p'	Observation
0 (black)	255 (white)	Total inversion
128 (medium gray)	127 (medium gray)	Central value
255 (white)	0 (black)	Total inversion

3.14.3.4 📦 Input and Output Specification (VPL)

Input:

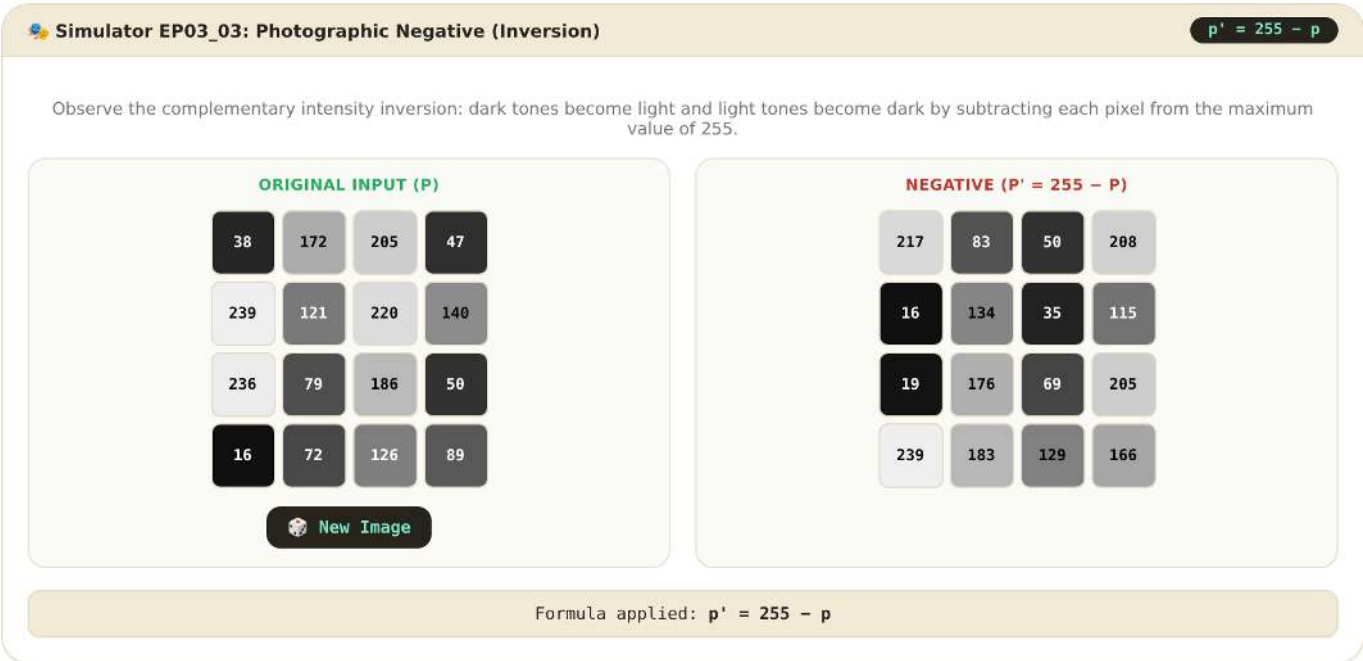
- Line 1: Integer L.
- Line 2: Integer C.
- Following lines: Integer elements of the original matrix.

Output:

- Negative matrix with L rows and C columns.

3.14.3.5 📌 Examples

Input	Output	Observation
140 128 200 255	255 127 55 0	Inversion of each pixel
2 2 10 20 30 40	245 235225 215	Inverted 2x2 matrix



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap03/sim-ep0303-inversao.html

Figure 3.28: EP03_03 Simulator: Image Inversion — Photographic Negative (p' = 255 − p)

```
%%writefile EP03_03.py
# Python code
```

```
Overwriting EP03_03.py
```

```
TestSuite("EP03_03.py").run()
```

```
<IPython.core.display.HTML object>
```

3.14.4 EP03_04 📊 Histogram Equalization (L bits)

In remote sensing satellite images, variation in illumination throughout the day produces low-contrast images. **Histogram equalization** is automatically applied in satellites such as Landsat to redistribute tones, revealing details of vegetation, relief, and urban areas that are invisible in the original image.

See Figure 3.29 for a simulation of this EP.

3.14.4.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows), C (columns), and B (number of bits, with $L_{\max} = 2^B$).
2. **Data:** Read the pixel matrix f with values in $[0, 2^B - 1]$.
3. **Histogram:** Compute $h[k]$ = number of pixels with intensity k, for $k = 0 \ldots 2^B - 1$.
4. **Probability:** $p[k] = h[k]/(L \cdot C)$.
5. **CDF:** $\text{cdf}[k] = \sum_{j=0}^{k} p[j]$; cumulative distribution function.
6. **LUT:** $\text{lut}[k] = \text{round}\left(\text{cdf}[k] \cdot (2^B - 1)\right)$; *Look-Up Table.*
7. **Application:** $g[i, j] = \text{lut}[f[i, j]]$.
8. **Output:** Display the equalized matrix of size $L \times C$.

3.14.4.2 📌 Computational Constraints

- **Rounding:** Use mathematical rounding (`round`) in the LUT.
- **Bits:** The number of levels is 2^B (e.g., $B = 3 \Rightarrow 8$ levels, $B = 8 \Rightarrow 256$ levels).
- **Cumulative CDF:** $\text{cdf}[k] = \sum_{j=0}^{k} p[j]$, with $\text{cdf}[2^B - 1] = 1.0$.

3.14.4.3 🧠 Theoretical Foundation

Step	Operation	Formula
1	Histogram	$h[k] \leftarrow$ number of pixels with intensity k
2	Probability	$p[k] = h[k]/(L \cdot C)$
3	CDF	$\text{cdf}[k] = \sum_{j=0}^{k} p[j]$
4	LUT	$\text{lut}[k] = \text{round}(\text{cdf}[k] \cdot (2^B - 1))$
5	Application	$g[i, j] = \text{lut}[f[i, j]]$

3.14.4.4 📦 Input and Output Specification (VPL)

Input:

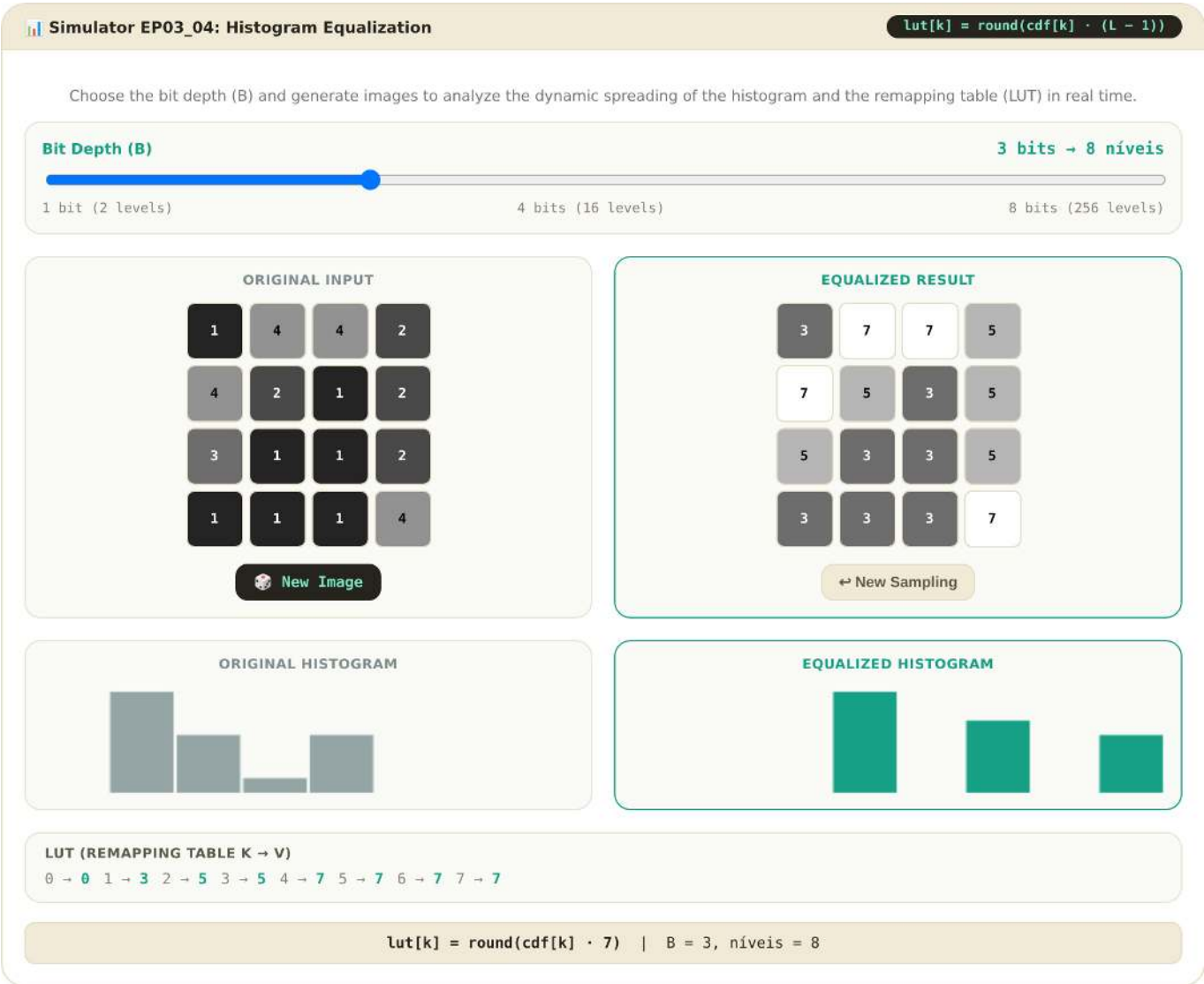
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer B (number of bits).
- Following lines: Integer elements of the matrix.

Output:

- Equalized matrix with L rows and C columns.

3.14.4.5 📌 Examples

Input	Output	Observation
5 5 3 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	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	Example with 3 bits from the chapter
1 4 3 0 0 7 7	0 0 7 7	Extreme bimodal histogram



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap03/sim-ep0304-equalizacao.html

Figure 3.29: Simulator EP03_04: Histogram Equalization (Levels L = 2^B)

```
%%writefile EP03_04.py
# Python code
```

```
Overwriting EP03_04.py
```

```
TestSuite("EP03_04.py").run()
```

```
<IPython.core.display.HTML object>
```

3.14.5 EP03_05 🔲 Binary AND Mask Application

In industrial computer vision inspection systems, it is necessary to isolate regions of interest (ROI) in part images to verify manufacturing defects. The **bitwise AND operation with a binary mask** is the fundamental mechanism for precisely cropping the inspection area, zeroing all pixels outside it.

See Figure 3.30 for a simulation of this exercise.

3.14.5.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Data:** Read the pixel matrix f (values $\in [0, 255]$).
3. **Mask:** Read the binary matrix m (values: only 0 or 255).
4. **Mapping:** For each pixel (i, j), apply the bitwise AND:

$$g(i, j) = f(i, j) \text{ AND } m(i, j)$$

where $255 =$ `11111111` and $0 =$ `00000000` in binary.

5. **Output:** Display the resulting $L \times C$ matrix.

3.14.5.2 📌 Computational Constraints

- **AND with 255:** p AND $255 = p$ (all bits preserved).
- **AND with 0:** p AND $0 = 0$ (all bits zeroed).
- **Mask:** The only possible values in the mask are 0 and 255.
- **Implementation:** In Python, bitwise AND between integers uses the `&` operator.

3.14.5.3 🧠 Theoretical Foundation

Pixel f	Mask m	Result f AND m
any v	255 (`11111111`)	v (preserved)
any v	0 (`00000000`)	0 (zeroed)

3.14.5.4 📦 Input and Output Specification (VPL)

Input:

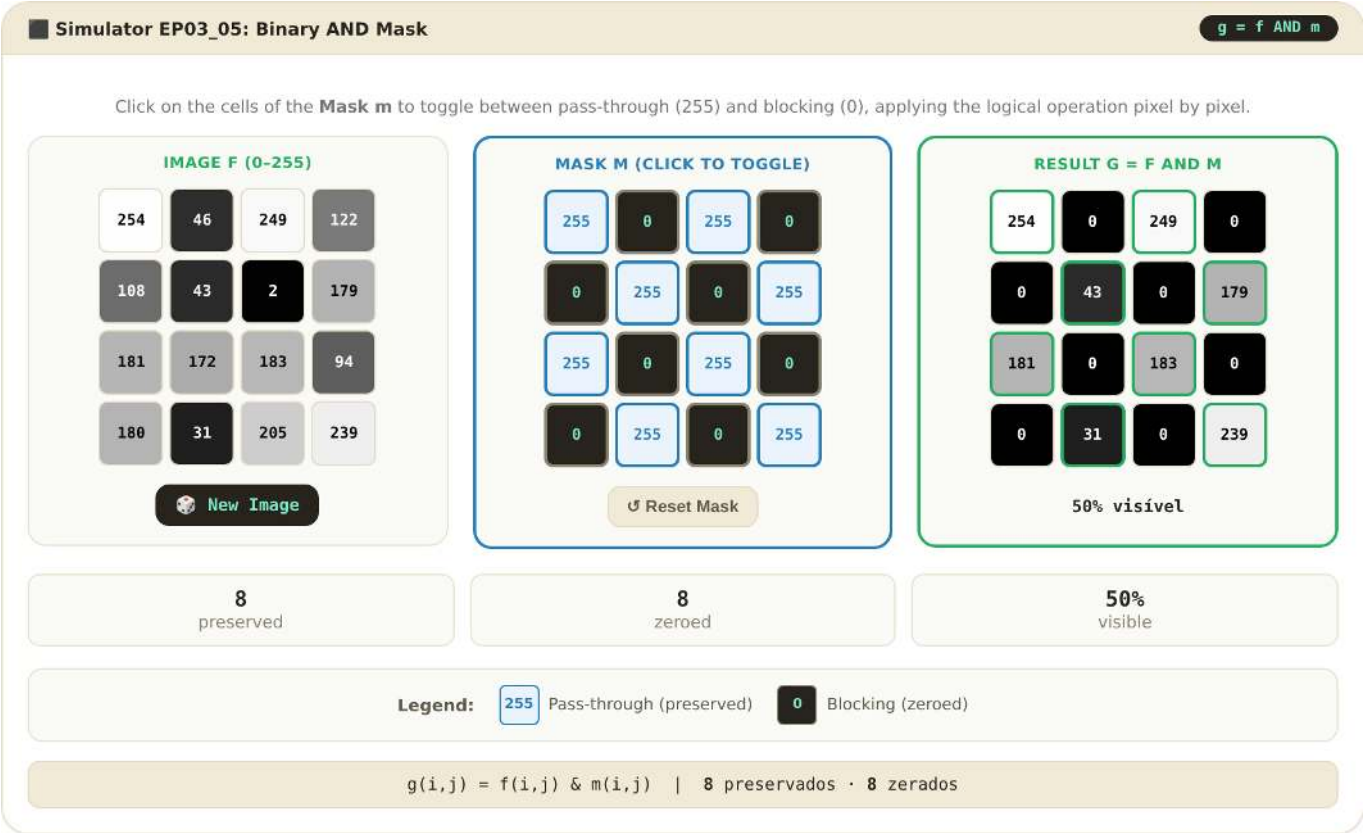
- Line 1: Integer L.
- Line 2: Integer C.
- Following lines: Elements of f (L rows).
- Following lines: Elements of m (L rows with values 0 or 255).

Output:

- Resulting $L \times C$ matrix.

3.14.5.5 📌 Examples

Input	Output	Observation
2 3 100 150 200 50 80 120 255 255 0 0 255 255	100 150 0 0 80 120	Mask selects region
1 4 10 20 30 40 255 0 255 0	10 0 30 0	Alternating preserved/zeroed



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap03/sim-ep0305-mascara.html

Figure 3.30: EP03_05 Simulator: Binary AND Mask Application

```
%%writefile EP03_05.py
# Python code
```

```
Overwriting EP03_05.py
```

```
TestSuite("EP03_05.py").run()
```

```
<IPython.core.display.HTML object>
```

3.14.6 EP03_06 N×N Mean Filter *Kernel*

In autonomous vehicle cameras, images captured under rain or fog exhibit Gaussian noise. The **mean filter** is widely used for real-time noise reduction, being implemented directly in the **ISP** (*Image Signal Processor*) of **CMOS** (*Complementary Metal-Oxide-Semiconductor*) sensors.

CMOS sensors are the image sensors used in most modern cameras (*smartphones*, *webcams*, automotive cameras, etc.). They convert light into electrical signals, and the ISP processes these signals in real time — applying operations such as noise reduction, white balance, and other image adjustments.

See the simulation of this EP in Figure 3.31.

3.14.6.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows), C (columns), and N (kernel size, always odd).
2. **Data:** Read the pixel matrix f.
3. **Mean Filter:** For each **internal** pixel (i, j) (without borders), compute:

$$g(i,j) = \text{round}\left(\frac{1}{N^2}\sum_{s=-(r)}^{r}\sum_{t=-(r)}^{r} f(i+s,\, j+t)\right), \quad r = \lfloor N/2 \rfloor$$

4. **Border Handling:** Pixels at the border (where the $N \times N$ window exceeds the limits) must be **copied directly** from the original without modification.
5. **Output:** Display the resulting $L \times C$ matrix.

3.14.6.2 📌 Computational Constraints

- **Radius:** $r = \lfloor N/2 \rfloor$ (half of the kernel, integer).
- **Internal pixels:** (i, j) with $r \leq i < L - r$ and $r \leq j < C - r$.
- **Rounding:** Use mathematical rounding before converting to integer.
- **No clipping:** The average of values $\in [0, 255]$ remains in $[0, 255]$.

3.14.6.3 🧠 Theoretical Background

Size N	Coefficient	Pixels in the window	Effect
3	$1/9 \approx 0.111$	9	Smooth
5	$1/25 = 0.04$	25	Medium
7	$1/49 \approx 0.020$	49	Strong

3.14.6.4 📦 Input and Output Specification (VPL)

Input:

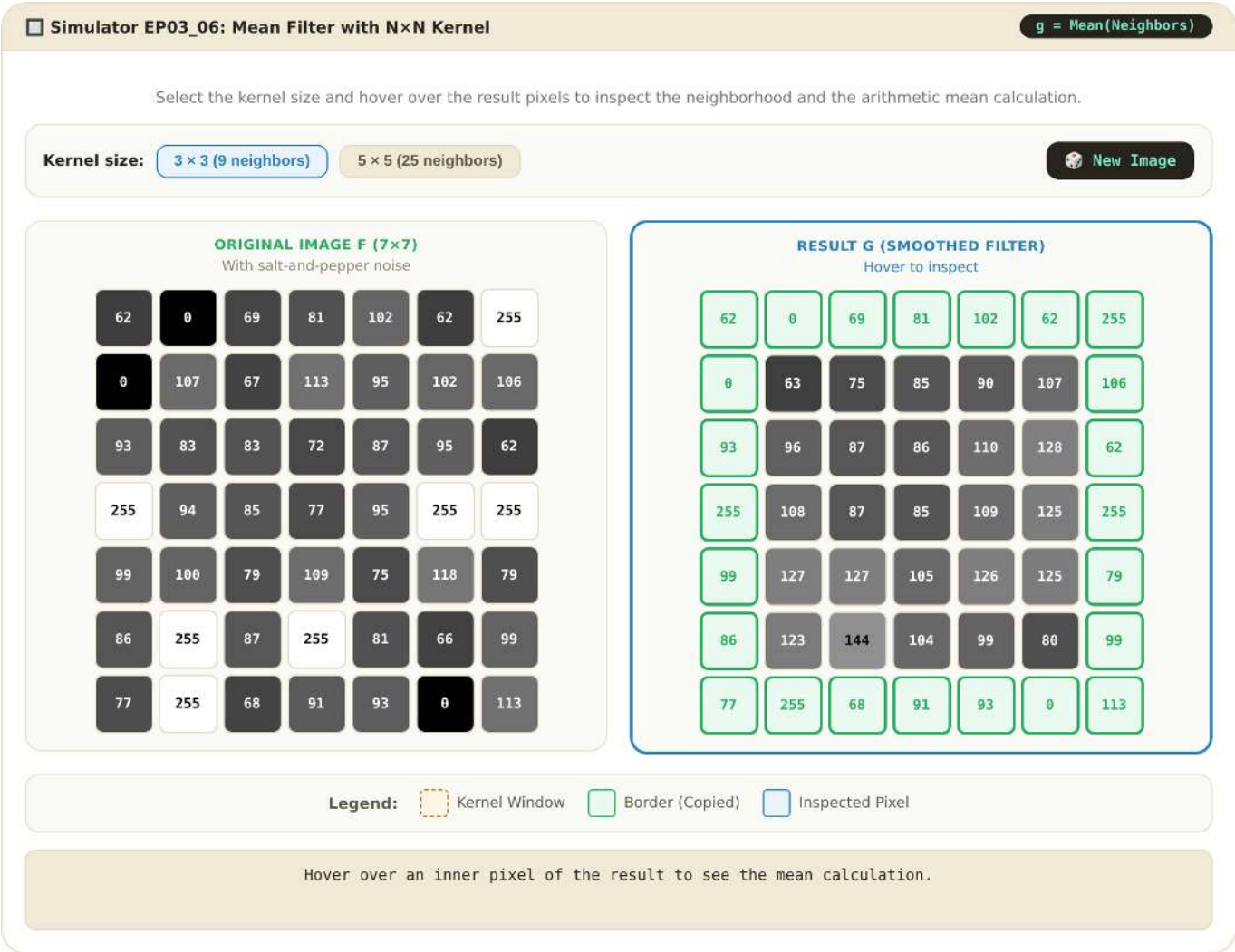
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer N (odd, $N \geq 3$).
- Following lines: Elements of the original matrix.

Output:

- Filtered $L \times C$ matrix.

3.14.6.5 📌 Examples

Input	Output	Observation
3 3 3 10 20 30 40 50 60 70 80 90	10 20 30 40 50 60 70 80 90	Only border (3×3 = all border)
5 5 3 0 0 0 0 0 0 0 0 0 0 0 0 100 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 11 11 11 0 0 11 11 11 0 0 11 11 11 0 0 0 0 0 0	Isolated pixel: all 9 internal pixels whose 3×3 window includes the value 100 receive round(100/9)=11



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap03/sim-ep0306-media.html

Figure 3.31: Simulator EP03_06: Average Filter with N×N Kernel

```
%%writefile EP03_06.py
# Python code
```

```
Overwriting EP03_06.py
```

```
TestSuite("EP03_06.py").run()
```

```
<IPython.core.display.HTML object>
```

3.14.7 EP03_07 🔍 Laplacian Operator (w4) for Edge Enhancement

In high-resolution tomography, the sharpness of edges between tissues is critical for diagnosis. The **Laplacian operator** is widely used in medical image preprocessing *pipelines* to automatically enhance anatomical contours before segmentation, avoiding manual intervention by the radiologist.

See Figure 3.32 for a simulation of this EP.

3.14.7.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Data:** Read the pixel matrix f.
3. **Laplacian (w4):** For each **internal** pixel (i, j) with $1 \leq i < L - 1$, $1 \leq j < C - 1$, compute:

$$\nabla^2 f(i,j) = f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1) - 4 \cdot f(i,j)$$

4. **Enhancement:** Compute the enhanced image:

$$g(i,j) = \text{clip}(f(i,j) - \nabla^2 f(i,j))$$

5. **Border:** Border pixels are copied directly: $g(i,j) = f(i,j)$.
6. **Output:** Display the enhanced matrix $L \times C$.

3.14.7.2 📌 Computational Constraints

- ***Kernel* w4:** $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ — only 4-neighbors.
- **Saturation:** $\text{clip}(x) = \max(0, \min(255, x))$ applied to the enhancement result.
- **No rounding:** The Laplacian uses only integer additions/subtractions.

3.14.7.3 🧠 Theoretical Background

Region	$\nabla^2 f$	Enhancement Effect
Uniform	≈ 0	No change
Rising edge	< 0	Pixel lightened
Falling edge	> 0	Pixel darkened

3.14.7.4 📦 Input and Output Specification (VPL)

Input:

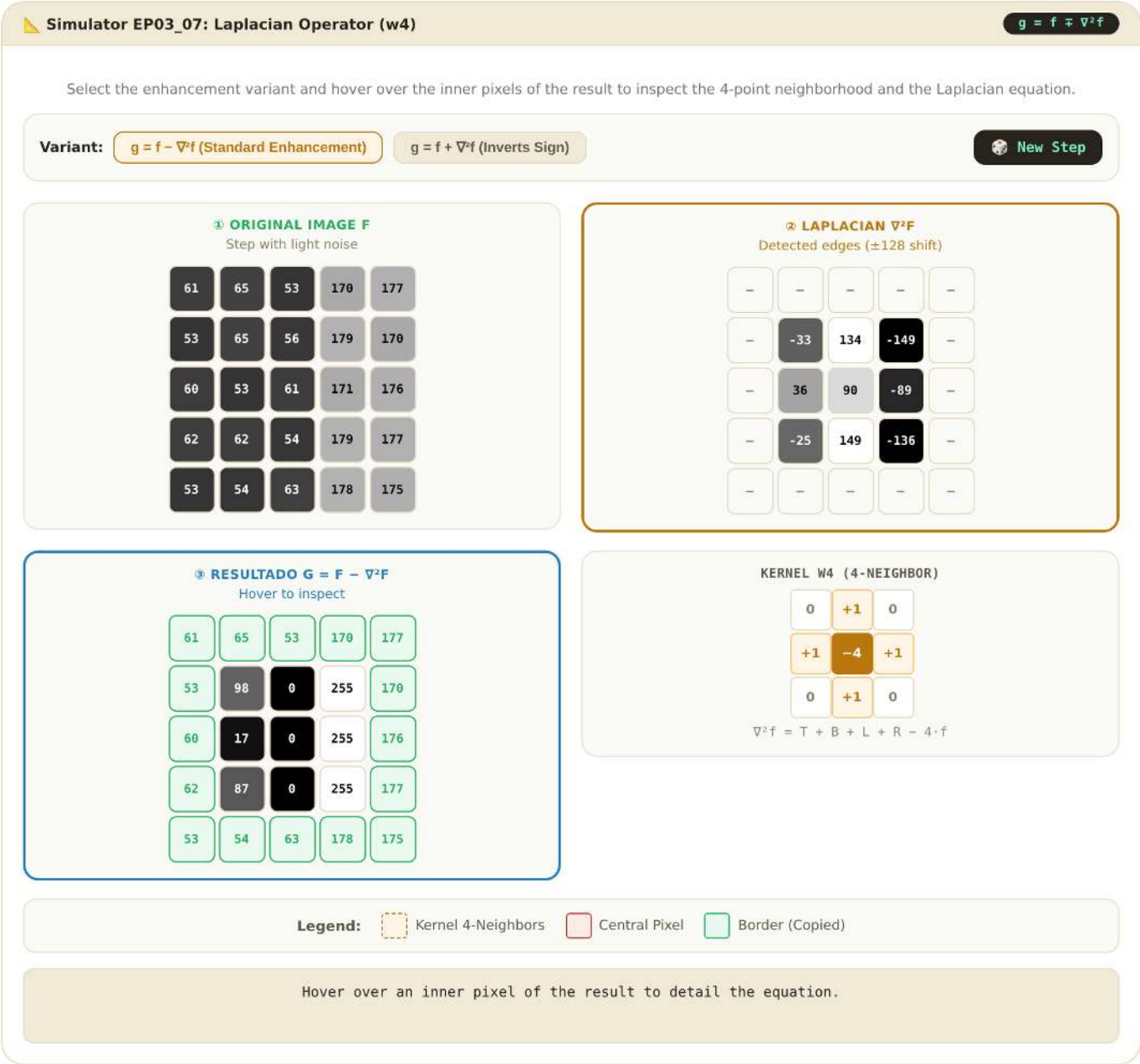
- Line 1: Integer L.
- Line 2: Integer C.
- Following lines: Elements of the original matrix.

Output:

- Enhanced matrix $L \times C$.

3.14.7.5 📌 Examples

Input	Output	Observation
3 3 0 0 0 0 100 0 0 0 0	0 0 0 0 255 0 0 0 0	Isolated peak: lap=−400, g=100−(−400)=500 → clip=255
3 3 50 50 50 50 50 50 50 50 50	50 50 50 50 50 50 50 50 50	Uniform region: Laplacian=0, no change



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap03/sim-ep0307-laplaciano.html

Figure 3.32: Simulator EP03_07: Laplacian Operator (w4) for Edge Enhancement

```
%%writefile EP03_07.py
# Python Code
```

```
Overwriting EP03_07.py
```

```
TestSuite("EP03_07.py").run()
```

```
<IPython.core.display.HTML object>
```

3.14.8 EP03_08 🧭 Sobel Gradient: Gx and Gy

In Mars exploration rovers (such as Perseverance), obstacle detection is performed in real time by stereoscopic cameras. The **Sobel operator** computes the directional gradient of the scene and is used in the edge detection algorithm to identify rocks, cracks, and terrain unevenness that could compromise navigation.

See Figure 3.33 for a simulation of this EP.

3.14.8.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Data:** Read the matrix f.
3. **Gx and Gy:** For each **internal** pixel (i, j) with $1 \leq i < L-1$, $1 \leq j < C-1$:

$$G_x(i,j) = [f(i-1,j+1) + 2f(i,j+1) + f(i+1,j+1)] - [f(i-1,j-1) + 2f(i,j-1) + f(i+1,j-1)]$$

$$G_y(i,j) = [f(i+1,j-1) + 2f(i+1,j) + f(i+1,j+1)] - [f(i-1,j-1) + 2f(i-1,j) + f(i-1,j+1)]$$

4. **Magnitude:** $|\nabla f(i,j)| = \text{clip}(\text{round}(\sqrt{G_x^2 + G_y^2}))$.
5. **Edge:** Edge pixels receive magnitude 0.
6. **Output:** Display the $L \times C$ magnitude.

3.14.8.2 📌 Computational Constraints

- **Rounding:** Apply `round` before converting to integer.
- **Saturation:** $\text{clip}(x) = \max(0, \min(255, x))$.
- **Square root:** Use $\sqrt{G_x^2 + G_y^2}$ (not the approximation $|G_x| + |G_y|$).

3.14.8.3 🧠 Theoretical Foundation

Operator	Detects	Diagonal coefficients
G_x	Vertical edges	± 1
G_y	Horizontal edges	± 1
$\|\nabla f\|$	All edges	Combined

3.14.8.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.

- Following lines: Matrix elements.

Output:

- Gradient magnitude, $L \times C$ matrix.

3.14.8.5 📌 Examples

Input	Output	Observation
3 3 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0	Null image: zero gradient
3 3 0 0 255 0 0 255 0 0 255	0 0 0 0 255 0 0 0 0	Central vertical edge: high Gx

```
%%writefile EP03_08.py
# Python code
```

```
Overwriting EP03_08.py
```

```
TestSuite("EP03_08.py").run()
```

```
<IPython.core.display.HTML object>
```

3.14.9 EP03_09 📡 3×3 Median Filter

Synthetic aperture radar (SAR) images used in environmental and military monitoring suffer from a specific type of noise called *speckle*, which has characteristics similar to salt-and-pepper noise. The **median filter** is the standard method for removing this noise because it preserves the edges of structures while eliminating spurious points.

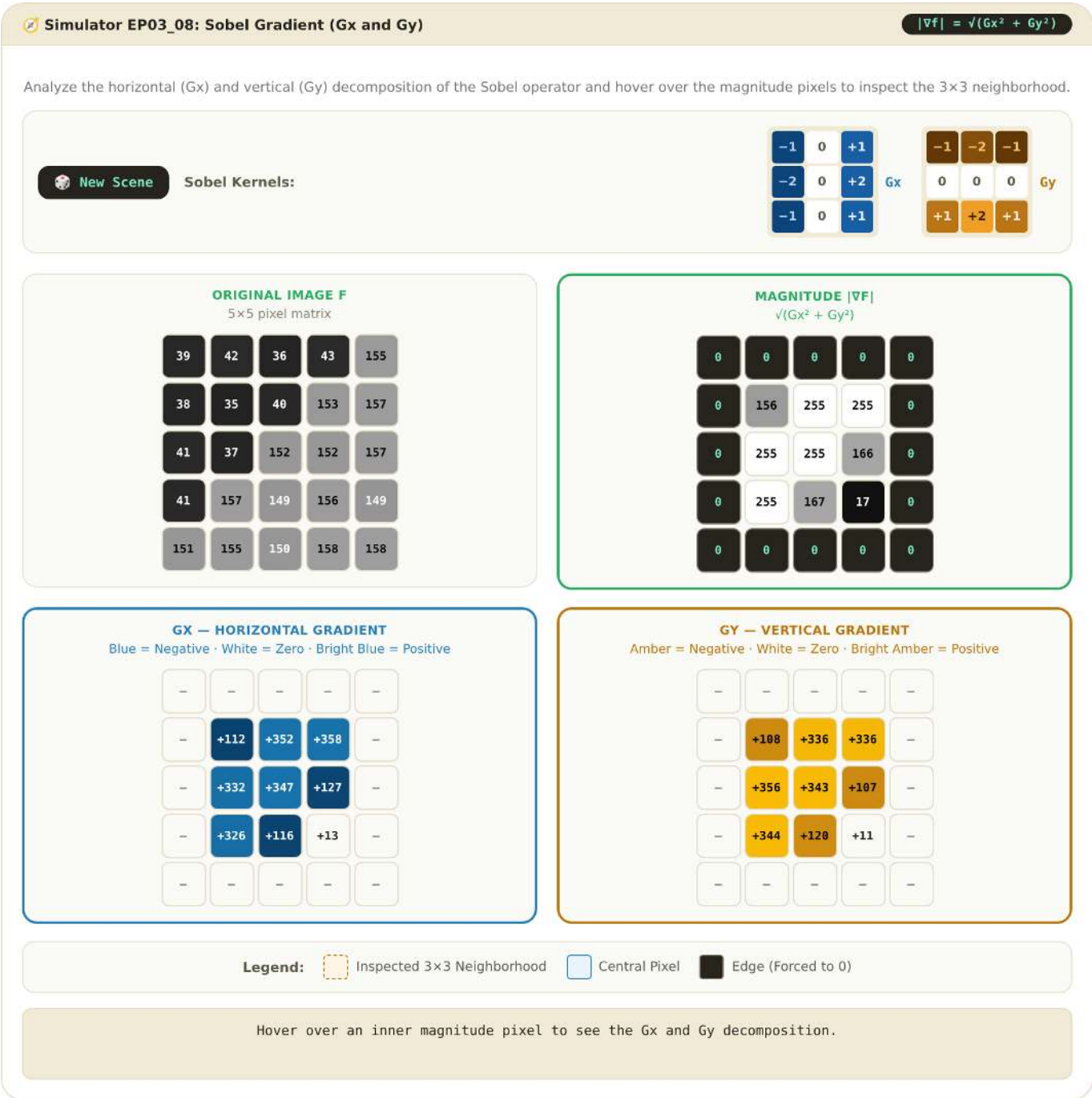
See Figure 3.34 for a simulation of this exercise.

3.14.9.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Data:** Read the pixel matrix f.
3. **3×3 Median Filter:** For each **internal** pixel (i, j) with $1 \leq i < L - 1$, $1 \leq j < C - 1$:
   - Collect the 9 pixels from the 3×3 neighborhood: $\{f(i + s, j + t) : s, t \in \{-1, 0, 1\}\}$.
   - Sort the 9 values in ascending order.
   - Assign $g(i, j)$ to the central value (index position 4, considering index 0).

$$g(i, j) = \text{median}\{f(i + s, j + t) : s, t \in \{-1, 0, 1\}\}$$

4. **Border:** Copy directly: $g(i, j) = f(i, j)$.
5. **Output:** Display the filtered $L \times C$ matrix.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap03/sim-ep0308-sobel.html

Figure 3.33: EP03_08 Simulator: Sobel Gradient (Gx and Gy)

3.14.9.2 📌 Computational Constraints

- **Window:** Always $3 \times 3 = 9$ elements.
- **Median:** The central element of the sorted sequence (index 4 from 0 to 8).
- **No clipping:** The median of values in $[0, 255]$ remains in $[0, 255]$.
- **Nonlinear:** The median filter cannot be expressed as a linear convolution.

3.14.9.3 🧠 Theoretical Background

Noise	Mean Filter	Median Filter
Salt and pepper (0 or 255)	Spreads the noise	Removes without distorting edges
Gaussian	Reduces effectively	Reduces partially

3.14.9.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Following lines: Matrix elements.

Output:

- Filtered $L \times C$ matrix.

3.14.9.5 📌 Examples

Input	Output	Observation
3 3 100 100 100 100 0 100 100 100 100	100 100 100 100 100 100 100 100 100	Black point removed: median of $8{\times}100{+}1{\times}0 = 100$
3 3 50 50 50 50 255 50 50 50 50	50 50 50 50 50 50 50 50 50	White point (salt) removed

```
%%writefile EP03_09.py
# Python code
```

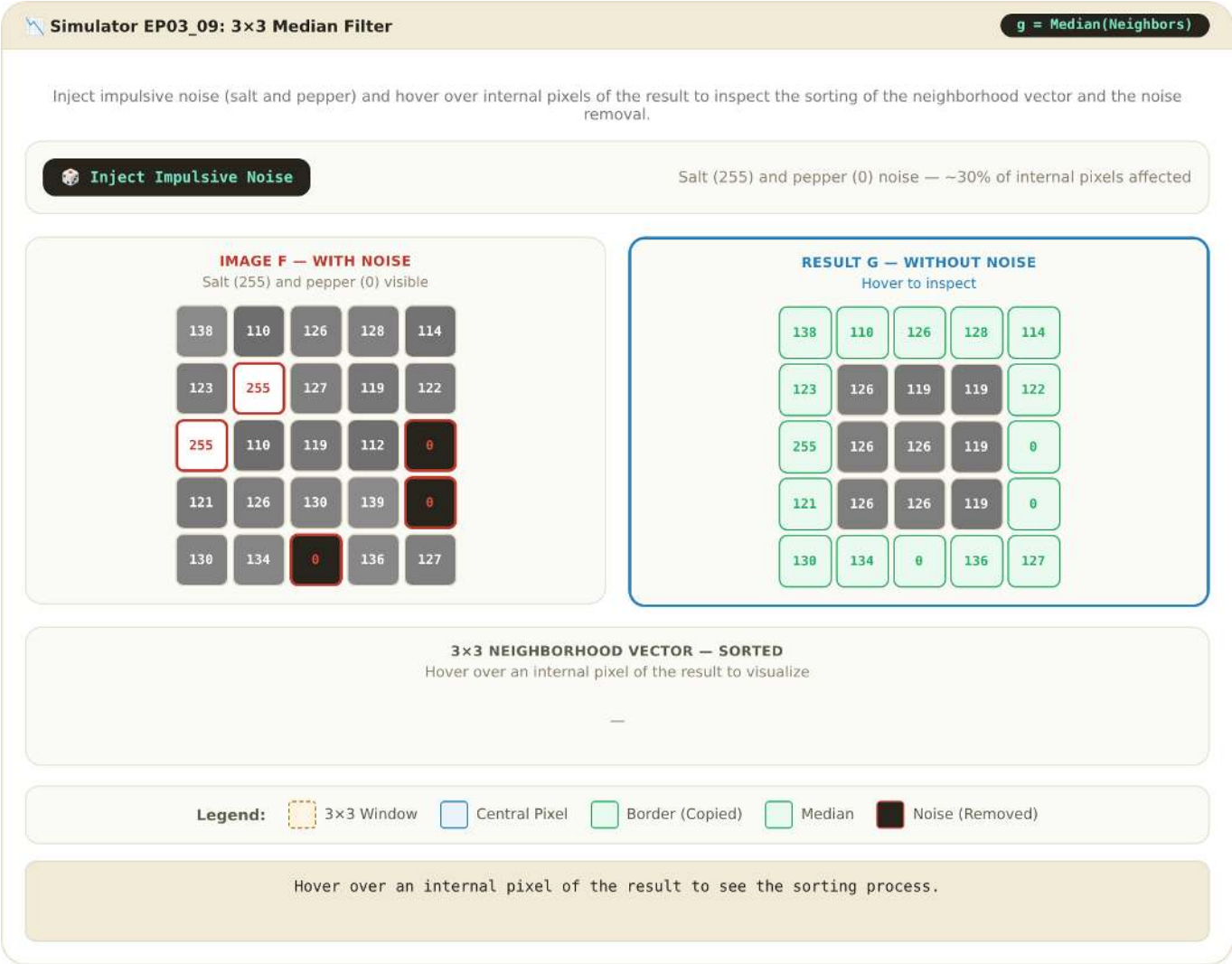
```
Overwriting EP03_09.py
```

```
TestSuite("EP03_09.py").run()
```

```
<IPython.core.display.HTML object>
```

3.14.10 EP03_10 ✨ *Unsharp Masking* (USM)

In digitization systems for historical documents and works of art, image sharpness is essential for reading handwritten texts and ornamental details. ***Unsharp Masking*** **(USM)** is the standard sharpening



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap03/sim-ep0309-mediana.html

Figure 3.34: EP03_09 Simulator: 3×3 Median Filter

enhancement algorithm used in professional scanners and software such as Adobe Photoshop, controlled by the parameter k that determines the enhancement intensity.

See Figure 3.35 for a simulation of this EP.

3.14.10.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Parameter:** Read the real value k (enhancement intensity, $k \geq 0$).
3. **Data:** Read the pixel matrix f.
4. **Smoothing:** Compute $\bar{f}$ with a 3×3 averaging filter (internal pixels only; borders preserved):

$$\bar{f}(i,j) = \frac{1}{9} \sum_{s=-1}^{1} \sum_{t=-1}^{1} f(i+s, j+t)$$

5. **High-frequency mask:** $m(i,j) = f(i,j) - \bar{f}(i,j)$.
6. **USM Enhancement:** For each internal pixel:

$$g(i,j) = \text{clip}\left(\text{round}\left(f(i,j) + k \cdot m(i,j)\right)\right)$$

7. **Border:** $g(i,j) = f(i,j)$ (direct copy).
8. **Output:** Display the enhanced matrix $L \times C$.

3.14.10.2 📌 Computational Constraints

- **Rounding:** Apply `round` before clipping.
- **Saturation:** $\text{clip}(x) = \max(0, \min(255, x))$.
- **Float operations:** Compute $\bar{f}$ and m in floating point before rounding the final result.
- $k = 0$**:** No enhancement — the output is identical to the input (except for borders).

3.14.10.3 🧠 Theoretical Foundation

Step	Operation	Description
1	$\bar{f} = f * \frac{1}{9}\mathbf{1}_{3\times 3}$	Smoothing (low frequencies)
2	$m = f - \bar{f}$	Mask (high frequencies)
3	$g = \text{clip}(\text{round}(f + k \cdot m))$	Weighted enhancement

3.14.10.4 📦 Input and Output Specification (VPL)

Input:

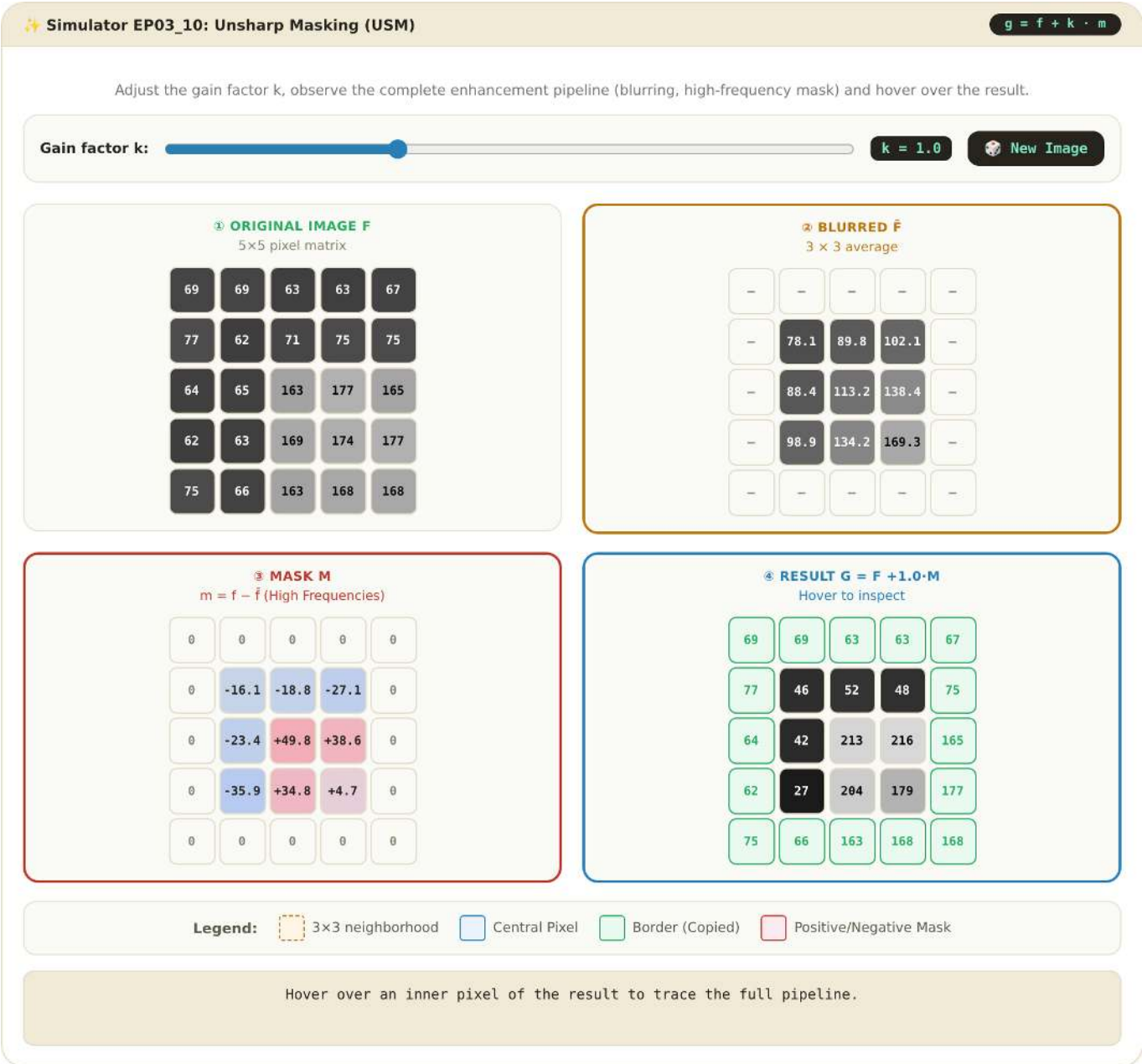
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Real k.
- Subsequent lines: Elements of the original matrix.

Output:

- Enhanced matrix $L \times C$.

3.14.10.5 📌 Examples

Input	Output	Observation
3	100 100 100	k=0: no enhancement
3	100 100 100	
0.0	100 100 100	
100 100 100		
100 100 100		
100 100 100		
3	50 50 50	k=1: center pixel enhanced and saturated
3	50 255 50	
1.0	50 50 50	
50 50 50		
50 200 50		
50 50 50		



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap03/sim-ep0310-unsharp.html

Figure 3.35: EP03_10 Simulator: *Unsharp Masking* (USM)

```
%%writefile EP03_10.py
# Python code
```

```
Overwriting EP03_10.py
```

```
TestSuite("EP03_10.py").run()
```

```
<IPython.core.display.HTML object>
```

Chapter 4

Mathematical Morphology and Image Segmentation

This chapter presents two fundamental topics in Digital Image Processing (DIP): **mathematical morphology** and **image segmentation**. Mathematical morphology provides a theoretical framework based on set theory for analyzing, refining, and quantifying the shape of objects in binary and grayscale images, through fundamental operators such as **erosion** and **dilation**. Segmentation, in turn, aims to partition the image into regions of interest, separating objects from the background and producing suitable representations for analysis and interpretation.

The chapter begins with **thresholding**, one of the most important segmentation techniques, introducing the automatic **Otsu** method and revisiting histogram analysis through interclass variance, as presented in Chapter 1. Next, the main mathematical morphology operators are studied, including erosion, dilation, opening, closing, and morphological reconstruction, which allow refining binary masks and preserving relevant object structures. Finally, region-based segmentation techniques are presented, such as **connected component labeling**, the **distance transform**, and the marker-controlled ***watershed*** algorithm, culminating in the extraction of geometric descriptors and the generation of *bounding boxes* compatible with modern object detection systems.

4.1 Objectives

By the end of this chapter, you will be able to:

- **Apply thresholding:** Understand Otsu's automatic criterion by maximizing inter-class variance (σ_B^2) and select appropriate preprocessing strategies to facilitate segmentation;
- **Master binary morphology:** Understand and apply erosion ($A \ominus B$) and dilation ($A \oplus B$) as fundamental operators, deriving opening ($A \circ B$), closing ($A \bullet B$), and operations based on morphological reconstruction, such as `mm.clohole` and `mm.edgeoff`;
- **Apply grayscale morphology:** Use morphological gradient and *top-hat* filters for enhancement and analysis of local structures;
- **Label connected components:** Identify and separate connected regions in binary images through labeling algorithms;
- **Apply distance transform:** Interpret and compute distances to the background using morphological approaches and geometric metrics;
- **Segment by regions:** Build marker-based segmentation *pipelines* using Distance Transform and the ***watershed*** algorithm;
- **Extract geometric descriptors:** Calculate properties such as area, perimeter, centroid, circularity, and *bounding boxes* through `mm.label0` and contour extraction;
- **Relate DIP and computer vision:** Understand how descriptors extracted by segmentation can be converted to formats used by modern detectors, such as YOLO.

```python
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup()
from morph import mm
import numpy as np
import cv2
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0
```

4.2 Thresholding

Thresholding is one of the simplest and most efficient forms of image segmentation. Its goal is to classify each pixel into two intensity classes, typically associated with *object* and *background*:

$$g(x,y) = \begin{cases} 255, & \text{if } f(x,y) > T \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

where $f(x,y)$ represents the pixel intensity in the original image and $g(x,y)$ is the resulting binary image.

The choice of the threshold T is important for the quality of the segmentation. The **Otsu** method automatically determines the optimal threshold by maximizing the **between-class variance** σ_B^2, defined by:

$$\sigma_B^2(T) = w_0(T)\, w_1(T)\, \left[\mu_0(T) - \mu_1(T)\right]^2 \tag{4.2}$$

where:

- $w_0(T)$ and $w_1(T)$ are the cumulative probabilities of the background and object classes;
- $\mu_0(T)$ and $\mu_1(T)$ are the mean intensities of these classes;
- $\sigma_B^2(T)$ represents the between-class variance for a given threshold T.

The method works best when the histogram presents two relatively separated groups of intensities. To that end, the algorithm evaluates all possible thresholds of the image — typically in the interval $[0, 255]$ for 8-bit images — and selects the value that maximizes the between-class variance, denoted by σ_B^2:

$$T^* = \arg \max_{T \in [0,255]} \sigma_B^2(T)$$

ℹ Otsu assumes bimodal histograms

The Otsu method yields better results when the histogram presents two well-defined peaks (*bimodality*), corresponding to the background and the object. The greater the separation between these peaks and the more pronounced the maximum of σ_B^2, the more reliable the obtained threshold tends to be.
In images with non-uniform illumination or multiple intensity regions, **adaptive thresholding** techniques — in which the threshold is computed locally — often produce more robust segmentations.
The subscript B in σ_B^2 stands for ***between classes***. Thus, σ_B^2 represents the **between-class variance**.

4.2.1 Image of Coins

The image used to practice segmentation is a photograph of a collection of coins from different countries and eras (Figure 4.1). Credit: GAZI.MD.AHAD (CC BY-SA 4.0). It features circular objects with well-defined edges, making it ideal for demonstrating thresholding, morphological operators, distance transform, *watershed*, and shape descriptors.

```
import os

url     = "https://upload.wikimedia.org/wikipedia/commons/2/25/GAZI.MD.AHAD_11.jpg"
caminho = "imagens/coins.png"

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

img_coins_color = mm.read(caminho)
img_coins_gray  = mm.gray(img_coins_color)

print(f"Dimensions [y,x,c]: {img_coins_color.shape}")
mm.show(img_coins_color, scale=30)
```

```
Dimensions [y,x,c]: (2560, 1920, 3)
```

Figure 4.1: Image with coins of various types. Credit: GAZI.MD.AHAD (CC BY-SA 4.0).

4.2.2 Preprocessing for Otsu

The quality of the Otsu method depends directly on how **bimodal** the histogram of the input image is. The original image of the coins exhibits non-uniform illumination and dark coins (oxidized copper) with intensities close to those of the dark background, making the histogram poorly bimodal.

To minimize these limitations, two classical preprocessing techniques, presented in the previous chapter, will be evaluated, applied before the thresholding step. Table 4.1 summarizes the characteristics of each

approach. These techniques aim to increase the separation between object and background, making the histogram closer to a bimodal distribution.

Table 4.1: Preprocessing techniques evaluated to improve the separation between coins and background before applying the Otsu method.

Technique	What it does	When to use
CLAHE	Local adaptive histogram equalization	Low global or regional contrast
Gaussian	Smoothing by convolution with a Gaussian	High-frequency noise (background texture)

The function `cv2.createCLAHE(clipLimit, tileGridSize)` divides the image into blocks and applies histogram equalization to each of them, limiting noise amplification through the `clipLimit` parameter. The Gaussian filter (`cv2.GaussianBlur`), in turn, smooths fine textures that could create spurious peaks in the histogram.

To objectively compare which preprocessing produces the best input for the Otsu method, the following will be presented for each version of the image: the image itself, the histogram with the optimal threshold T^* highlighted, the curve $\sigma_B^2(T)$, and the binarization result.

Comparison criterion

The version that exhibits the highest value of σ_B^2 at its peak provides the best separation between the background and object classes and, consequently, the best input for the Otsu method (Figure 4.2).

```
import cv2, io
import matplotlib.pyplot as plt

def otsu_criterio(img):
    h=mm.hist(img); p=h/h.sum();                  # histogram and probabilities
    sigma2=np.zeros(len(p))
    for T in range(1,len(p)):                     # loop over thresholds
        w0,w1=p[:T].sum(),p[T:].sum()             # class probabilities
        if w0*w1==0: continue                     # avoid division by zero
        mu0=(np.arange(T)*p[:T]).sum()/w0         # background mean
        mu1=(np.arange(T,len(p))*p[T:]).sum()/w1  # object mean
        sigma2[T]=w0*w1*(mu0-mu1)**2              #  ²B(T)
    return sigma2,np.argmax(sigma2)               # curve and optimal T

def fig2img(fig):
    b=io.BytesIO(); fig.savefig(b,format='png',dpi=100)   # figure → 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}")       # optimal threshold
    ax.set(xlabel="T" if ylabel=="  ²B" else "Intensidade",ylabel=ylabel)
    ax.legend(fontsize=8); plt.tight_layout()
    return fig2img(fig)

#   Preprocessings
img_clahe = mm.clahe(img_coins_gray, clipLimit=2.0, tiles=8)
img_gauss = cv2.GaussianBlur(img_clahe,(5,5),0)
imgs0=[("Original",img_coins_gray),
```

```
        ("CLAHE",img_clahe),
        ("CLAHE+Gauss",img_gauss)]

#   Comparative table
print(f"{'Versão':<18}{'T*':>6}{' ²B pico':>14}")
print("-"*40)

imgs,titles=[],[]
for nome,img in imgs0:
    sigma2,T=otsu_criterio(img)                          # computes  ²B(T)
    print(f"{nome:<18}{T:>6}{sigma2[T]:>14.4e}")
    imgs += [
        img,                                             # image
        plot_curve(mm.hist(img),T,f"Hist T*={T}","Freq.","steelblue"),
        plot_curve(sigma2,T," ²B(T)"," ²B","darkorange"),
        mm.threshold(img)                               # final Otsu
    ]
    titles += [nome,f"Hist T*={T}"," ²B(T)",f"Otsu T*={T}"]

#   Final display
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
```

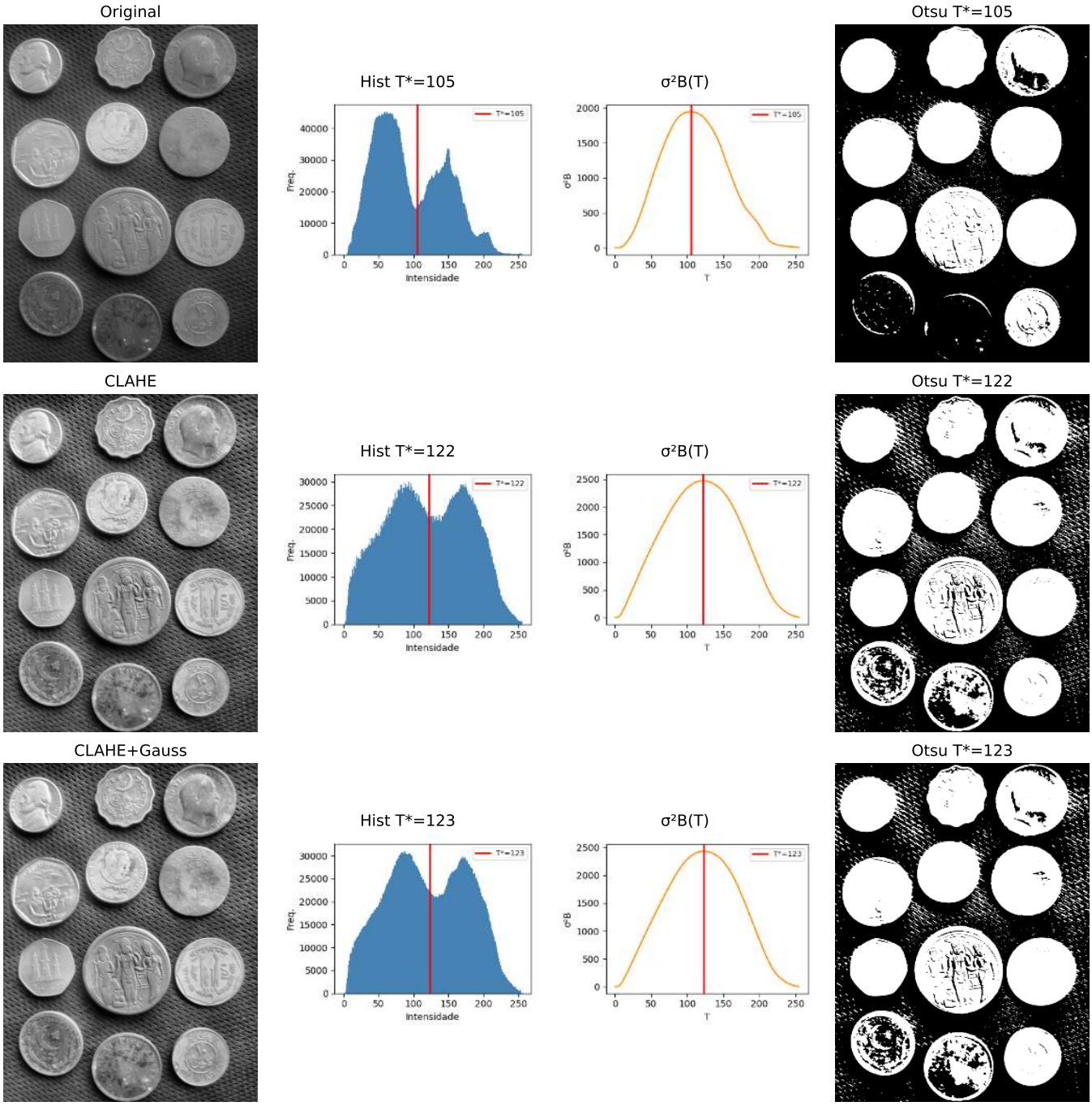


Figure 4.2: Comparison of preprocessings: image | histogram+T* | ²B(T) | Otsu. The best bimodal separation indicates the most reliable threshold.

4.2.3 Result: CLAHE as the Best Pre-processing

The analysis of Figure 4.2 indicates that **CLAHE** achieved the highest inter-class variance ($\sigma_B^2 \approx 2{,}47 \times 10^3$), with an optimal threshold $T^* = 122$. Although the **CLAHE+Gaussian** combination produced a very similar result ($\sigma_B^2 \approx 2{,}43 \times 10^3$, $T^* = 123$), the quantitative criterion of Otsu's method slightly favors the use of CLAHE alone.

In visual terms, the binarized images obtained with CLAHE and CLAHE+Gaussian are practically equivalent. The difference between the two approaches becomes more evident in the analysis of the histograms and the values of $\sigma_B^2(T)$ than in the direct inspection of the resulting segmentations. Thus, the choice of CLAHE is mainly based on maximizing the statistical separation between the background and object classes.

Interpretation of the results

Note that the CLAHE and CLAHE+Gaussian pre-processings produce very similar histograms and optimal thresholds ($T^* = 122$ and $T^* = 123$). Consequently, the resulting binarized images are also quite similar. In this case, the decision is not based on striking visual differences but on the objective criterion of Otsu's method: the highest value of σ_B^2 indicates the best separation between the classes.

4.3 Mathematical Morphology

Mathematical morphology is a set-based theory used to analyze the shape and structure of objects in images. Unlike the linear filters presented in Chapter 3, morphological operators are **nonlinear**, as they are based on minimum, maximum, and spatial inclusion operations, rather than linear combinations of intensities. These operators act on the neighborhood of each pixel through a **structuring element** $\mathbb{B}$, which defines the shape and size of the analyzed region.

In binary and grayscale images with flat elements, the structuring element translated to position x is defined spatially as:

$$\mathbb{B}_x = \{x + b \mid b \in \mathbb{B}\}$$

At image border regions, part of the set $\mathbb{B}_x$ may extend beyond the physical domain of the scene ($\mathbb{E}$). To ensure the mathematical consistency of the primitive operators at these boundaries, it is theoretically assumed that the space exterior to the image domain is filled with the **neutral element** of the corresponding operation (positive infinity for erosion and negative infinity for dilation), preventing the external environment from corrupting the internal structures of the object.

When the structuring element associates weights to its elements — that is, $b : \mathbb{B} \to \mathbb{Z}$ — it is termed a **structuring function** or **non-flat structuring element**.

Developed by Matheron and Serra in the 1960s for binary images and subsequently extended to grayscale, mathematical morphology underpins operators such as morphological gradient, *top-hat*, *watershed*, and distance transform, all derived from two primitives: **erosion** and **dilation** [Matheron (1975); Serra (1982)].

4.3.1 Erosion and Dilation

The two primitive operators are defined in a unified manner for grayscale images ($f : \mathbb{E} \to \mathbb{Z}$) and, by restriction to the domain $\{0, 1\}$, also for binary images.

4.3.1.1 Erosion

The **erosion** of an image f by a structuring function $b : \mathbb{B} \to \mathbb{Z}$ is formally defined by:

$$\varepsilon_b(f)(x) = (f \ominus b)(x) = \min_{z \in \mathbb{B}} \{ f(x + z) - b(z) \}, \quad \forall\, x \in \mathbb{E} \tag{4.3}$$

In practice, erosion replaces the intensity of pixel x with the minimum value resulting from the difference between the image and the structuring element in the neighborhood defined by the domain $\mathbb{B}$. Positive values in the weights of $b(z)$ force the local result downward, "digging" the image relief more deeply and intensifying the erosion.

In the **flat** case (where the weights are null within the domain, i.e., $b \equiv 0$), the expression simplifies to the pure local minimum:

$$\varepsilon_B(f)(x) = \min\{ f(y) : y \in \mathbb{B}_x \}$$

In binary images, this operation is equivalent to requiring that the set $\mathbb{B}$, translated to coordinate x, be **completely contained** in the object A:

$$A \ominus \mathbb{B} = \{ z \in \mathbb{E} \mid \mathbb{B}_z \subseteq A \}$$

Visual Effect: *Shrinks* bright objects and structures, eliminating protrusions, bright peaks, or noise that are geometrically smaller than the domain $\mathbb{B}$.

4.3.1.2 Implementation of erosion

The didactic version `mm.ero0` implements the particular case of erosion with a **flat structuring element**. For each pixel (y, x), the function traverses the spatial neighbors allowed by B and stores the smallest value found in the input image f, directly reproducing the local minimum operation described in Equation 4.3 for $b \equiv 0$.

Note that the neighbor values are always read statically from the original image f; the output matrix g is used exclusively to record the accumulated minimum of the current neighborhood. Thus, the final result is invariant with respect to the pixel scanning order (whether by rows or columns).

The auxiliary function `_viz` computes the coordinates of valid neighbors within the physical boundaries of the image. At the borders, the initialization of the accumulator to `255` exactly emulates the padding with the neutral element required by the theory. The interface function `mm.ero`, in turn, resorts to the native and optimized OpenCV implementation (`mm.ero`) when the structuring element is flat, switching to the general routine `mm.ero1` if the element has topographic weights.

The following computational example illustrates the application of a cross-shaped structuring element (`mm.secross()`) highlighted in Figure 4.3, comparing the execution of the didactic loop-based variant (`mm.ero0`) with the computational engine of OpenCV (`mm.ero`).

```
B_cruz = mm.secross()
mm.drawImgPlt(B_cruz, scale=40)
```

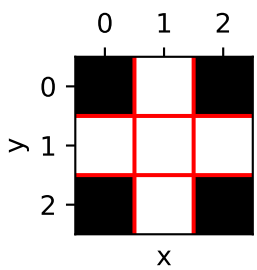


Figure 4.3: Structuring element in the shape of a cross (B_{cross}) used for 4-connectivity.

```
def _viz(f, B, y, x):
    """Generates (vy, vx, b_val) for each valid neighbor of (y,x)."""
    H, W = f.shape
    Bh, Bw = B.shape
    oh, ow = -Bh/2 + 0.5, -Bw/2 + 0.5  # fixed offsets
    for by, bx in np.ndindex(Bh, Bw):
        vy, vx = int(y + by + oh), int(x + bx + ow)
        if 0 <= vy < H and 0 <= vx < W:
            yield vy, vx, B[by, bx]

def ero(f, Bc=np.zeros((3,3),dtype='uint8')):
    """Erosion (OpenCV or with weights)."""
    try:    return cv2.erode(f, Bc)
```

```
    except: return mm.ero1(f, Bc)

def ero0(f, Bc=np.ones((3,3),dtype='uint8')):
    """Classical erosion without weights."""
    g = np.empty_like(f)
    for y in range(f.shape[0]):
        for x in range(f.shape[1]):
            g[y,x] = 255
            for vy,vx,bv in mm._viz(f,Bc,y,x):
                if bv and g[y,x] > f[vy,vx]: g[y,x] = f[vy,vx]
    return g

# Test and Validation Script
B = mm.secross()
print("Structuring element B:")
print(mm.drawImage(B))

f = mm.randomImage(5,5)
print("Original image f:")
print(mm.drawImage(f))

print("Didactic Flat Erosion (ero0):")
print(mm.drawImage(ero0(f, B)))

print("Optimized OpenCV Erosion (ero):")
print(mm.drawImage(ero(f, B)))
```

```
Structuring element B:
0 1 0
1 1 1
0 1 0

Original image f:
0 3 5 4 6
7 3 2 9 7
1 0 5 6 6
9 0 6 5 7
7 3 9 7 0

Didactic Flat Erosion (ero0):
0 0 2 4 4
0 0 2 2 6
0 0 0 5 6
0 0 0 5 0
3 0 3 0 0

Optimized OpenCV Erosion (ero):
0 0 2 4 4
0 0 2 2 6
0 0 0 5 6
0 0 0 5 0
3 0 3 0 0
```

The `_viz` function uses `yield` to generate each neighbor on demand, without storing all results in memory in a list. In the experiment below, a 3000×3000 window produces 9 million neighbors. The list-based implementation consumed more than 1 GB of RAM and took approximately 46 s, while the version with `yield` consumed negligible memory and ran in 38 s. In image processing, generators are especially useful for efficiently traversing large neighborhoods.

```python
import tracemalloc

def lista(n): return [(i, i) for i in range(n)]
def gera(n):
    for i in range(n): yield i, i

for nome, f in [("LISTA", lista), ("YIELD", gera)]:
    tracemalloc.start()
    sum(x+y for x,y in f(9_000_000))
    _, pico = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    print(f"{nome}: {pico/1024/1024:.1f} MB")
```

```
LISTA: 830.8 MB
YIELD: 0.0 MB
```

4.3.1.3 Dilation

Dilation of an image f by a structuring function $b : \mathbb{B} \to \mathbb{Z}$ is formally defined by:

$$\delta_b(f)(x) = (f \oplus b)(x) = \max_{z \in \mathbb{B}} \{ f(x - z) + b(z) \}, \quad \forall\, x \in \mathbb{E} \tag{4.4}$$

In practice, dilation replaces the intensity of pixel x with the largest value resulting from the sum between the image and the structuring element within the defined neighborhood. The spatial inversion argument $(x - z)$ indicates that dilation implicitly evaluates the transposed (reflected) element $\hat{b}$, a fundamental property for ensuring mathematical duality with respect to erosion.

In the **flat** case (where the weights are null within the domain, i.e., $b \equiv 0$), the expression reduces to pure local maximum:

$$\delta_B(f)(x) = \max\{ f(y) : y \in \mathbb{B}_x \}$$

For binary images, this operation is equivalent to requiring that the reflected set $\hat{\mathbb{B}}$, translated to coordinate x, has a **non-empty intersection** with object A:

$$A \oplus \mathbb{B} = \{ z \in \mathbb{E} \mid \hat{\mathbb{B}}_z \cap A \neq \emptyset \}$$

Visual Effect: *Expands* the bright structures of the image, increasing object filling, connecting nearby components, and eliminating channels, dark pits, or valleys that are geometrically smaller than the domain $\mathbb{B}$.

4.3.1.4 Implementation of dilation

The didactic version `mm.dil0` implements the particular case of dilation with a **flat structuring element**. For each pixel (y, x), the function traverses the spatial neighbors allowed by B and stores the largest value found in the input image f, directly reproducing the local maximum operation for $b \equiv 0$.

Before starting the spatial scan, the structuring element undergoes a geometric reflection through the `np.flip(Bc)` instruction to explicitly construct the transposed matrix $\hat{B}$ required by the theory. In perfectly symmetric masks (such as crosses, squares, and disks centered at the origin), this reflection does not alter the pixel arrangement; however, for asymmetric elements, this step is strictly necessary to ensure equivalence with the formal definitions and to safeguard the duality laws.

As verified in the erosion operator, neighbor values are always read statically from the original matrix f, while the output matrix g acts purely as the register of the accumulated neighborhood maximum. At the

image boundaries, initializing the accumulator to `0` exactly emulates the external padding with the neutral element ($-\infty$, or zero in 8-bit representations), ensuring that the physical edges of the scene are dilated in perfect conformity with the standard adopted by OpenCV.

The computational example below illustrates the practical application of a cross-shaped element (`mm.secross()`), validating the consistency between the loop-based logic (`mm.dil0`) and the native industrial method (`mm.dil`).

```
def dil(f, Bc=np.zeros((3,3),dtype='uint8')):
    """Dilation (OpenCV or with weights)."""
    try:    return cv2.dilate(f, Bc)
    except: return mm.dil1(f, Bc)

def dil0(f, Bc=np.zeros((3,3),dtype='uint8')):
    """Flat dilation strictly following the theory."""
    g = np.empty_like(f)
    Bc = np.flip(Bc)     # explicit reflection: B
    for y in range(f.shape[0]):
        for x in range(f.shape[1]):
            g[y,x] = 0 # Initializes with the minimum value to search for the maximum
            for vy,vx,bv in mm._viz(f,Bc,y,x):
                if bv and g[y,x] < f[vy,vx]:
                    g[y,x] = f[vy,vx]
    return g

# Test and Validation Script
B = mm.secross()
print("Structuring element B:")
print(mm.drawImage(B))

f = mm.randomImage(5,5)
print("Original image f:")
print(mm.drawImage(f))

print("Didactic Flat Dilation (dil0):")
print(mm.drawImage(dil0(f, B)))

print("Optimized OpenCV Dilation (dil):")
print(mm.drawImage(dil(f, B)))
```

```
Structuring element B:
0 1 0
1 1 1
0 1 0

Original image f:
1 7 7 9 6
4 8 5 3 6
7 4 8 2 9
8 7 0 1 4
4 7 2 4 2

Didactic Flat Dilation (dil0):
7 8 9 9 9
8 8 8 9 9
8 8 8 9 9
8 8 8 4 9
8 7 7 4 4

Optimized OpenCV Dilation (dil):
```

```
7 8 9 9 9
8 8 8 9 9
8 8 8 9 9
8 8 8 4 9
8 7 7 4 4
```

Note: The Sign Confrontation ($f(x+z)$ vs $f(x-z)$)

Compare the formal definitions of erosion (Equation 4.3) and dilation (Equation 4.4). Consider an asymmetric structuring element to the right $\mathbb{B} = \{0, 1\}$ (origin and one pixel to the right) applied at position $x = 10$.

1. **In Erosion** (Equation 4.3):

$$\min\{f(x+z) - b(z)\}$$

   - $z = 0 \Rightarrow f(10+0) = \mathbf{f(10)}$
   - $z = 1 \Rightarrow f(10+1) = \mathbf{f(11)}$

   The operator queries the current pixel (10) and the pixel to the right (11), preserving the original orientation of $\mathbb{B}$.

2. **In Dilation** (Equation 4.4):

$$\max\{f(x-z) + b(z)\}$$

   - $z = 0 \Rightarrow f(10-0) = \mathbf{f(10)}$
   - $z = 1 \Rightarrow f(10-1) = \mathbf{f(9)}$

   Due to the negative sign ($-z$), advancing in the structuring element corresponds to moving backward in the image, causing dilation to query the pixel to the left (9).

The `_viz` function, used in `morph.py`, generates neighbors by additive shifts of the form $x + z$. For this reason, the implementation of `mm.dil0` previously reflects the structuring element using `np.flip(B)`. After reflection, the scan based on $x + z$ accesses exactly the same points defined by the theoretical expression $f(x - z)$ of dilation in Equation 4.4.

For symmetric structuring elements (such as discs, squares, and centered crosses), reflection does not alter the mask. For asymmetric elements, however, this step is essential for the implementation to correctly reproduce the mathematical definition of dilation and preserve the erosion–dilation duality.

Erosion–dilation duality

Erosion and dilation are **dual by complement**. This means that one operator can be fully obtained from the other, provided that one operates on the complement of the image using the reflected structuring element $\hat{B}$:

$$(A \ominus B)^c = A^c \oplus \hat{B} \quad \Longleftrightarrow \quad A \ominus B = (A^c \oplus \hat{B})^c$$

Similarly, **dilation can also be obtained from erosion**:

$$(A \oplus B)^c = A^c \ominus \hat{B} \quad \Longleftrightarrow \quad A \oplus B = (A^c \ominus \hat{B})^c$$

In practical terms, the erosion of an object can be obtained by dilating its complement, followed by complementing the result (and vice versa). In the implementation of the `morph.py` package, the didactic versions `mm.ero0` and `mm.dil0` make this structure explicit through loops, while `mm.ero` and `mm.dil` delegate the operations to OpenCV for greater computational efficiency.

Boundary conditions and finite images

In classical mathematical morphology, defined over an infinite domain (typically $\mathbb{Z}^2$), this duality is exact. In digital images, however, one works with finite matrices, and the result depends on how pixels located outside the image are treated.
For the duality identities to remain valid, the complement must be defined relative to the same universe, and the boundary conditions adopted for erosion and dilation must be complementary to each other. For example, if erosion assumes that external pixels belong to the object (255), then dilation applied to the complement must assume that these same external pixels belong to the background (0).
When different filling strategies are used (replication, reflection, constant value, etc.), the theoretical duality may no longer be satisfied exactly in regions near the image borders.

To numerically illustrate the morphological operators and the erosion–dilation duality, Figure 4.4 presents a 10×10 binary image processed with an "L"-shaped structuring element. In the implementation of `morph.py`, the origin of B is set at the geometric center of the mask — position $(1, 1)$ for a 3×3 *kernel* — and must correspond to an active element for the erosion to behave correctly (as discussed earlier). The structuring element B_L defined below satisfies this condition. Figure 4.5 complements the analysis with an interactive erosion simulator, allowing visualization of the displacement of the structuring element over the image and identification of the positions where it remains completely contained within the object.

```
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)   # 'L', origin at the center
B_hat = np.array([[0,1,1],[0,1,1],[0,0,1]], dtype=np.uint8)   # B_L rotated 180° (B)

print("Structuring element B_L:"); print(mm.drawImg(B_L))
print("Reflected structuring element B:"); print(mm.drawImg(B_hat))

img_ero0 = mm.ero0(A, B_L)

# Duality: (A   B)  == A   B
A_c    = mm.neg(A)
ero_c  = mm.neg(img_ero0)
dil_Ac = mm.dil0(A_c, B_hat)
print(f"Duality (A   B)  == A   B : {np.array_equal(ero_c, dil_Ac)}")

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)
)
```

```
Structuring element B_L:
1 0 0
1 1 0
1 1 0

Reflected structuring element B:
```

```
0 1 1
0 1 1
0 0 1

Duality (A   B) == A   B : True
```

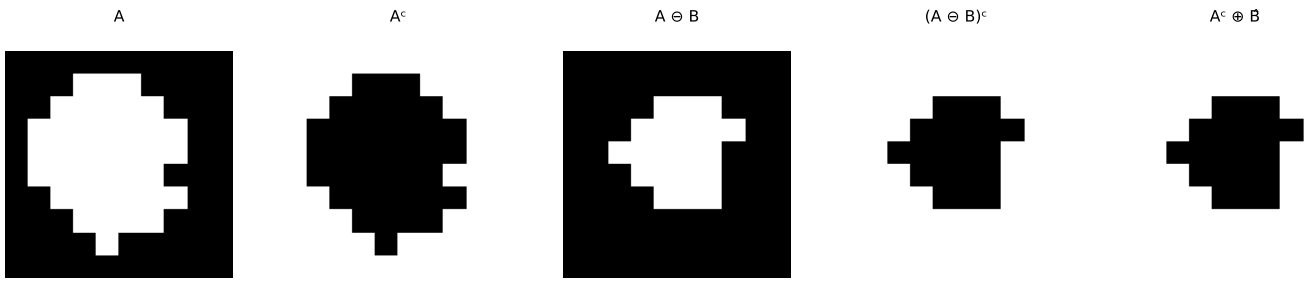
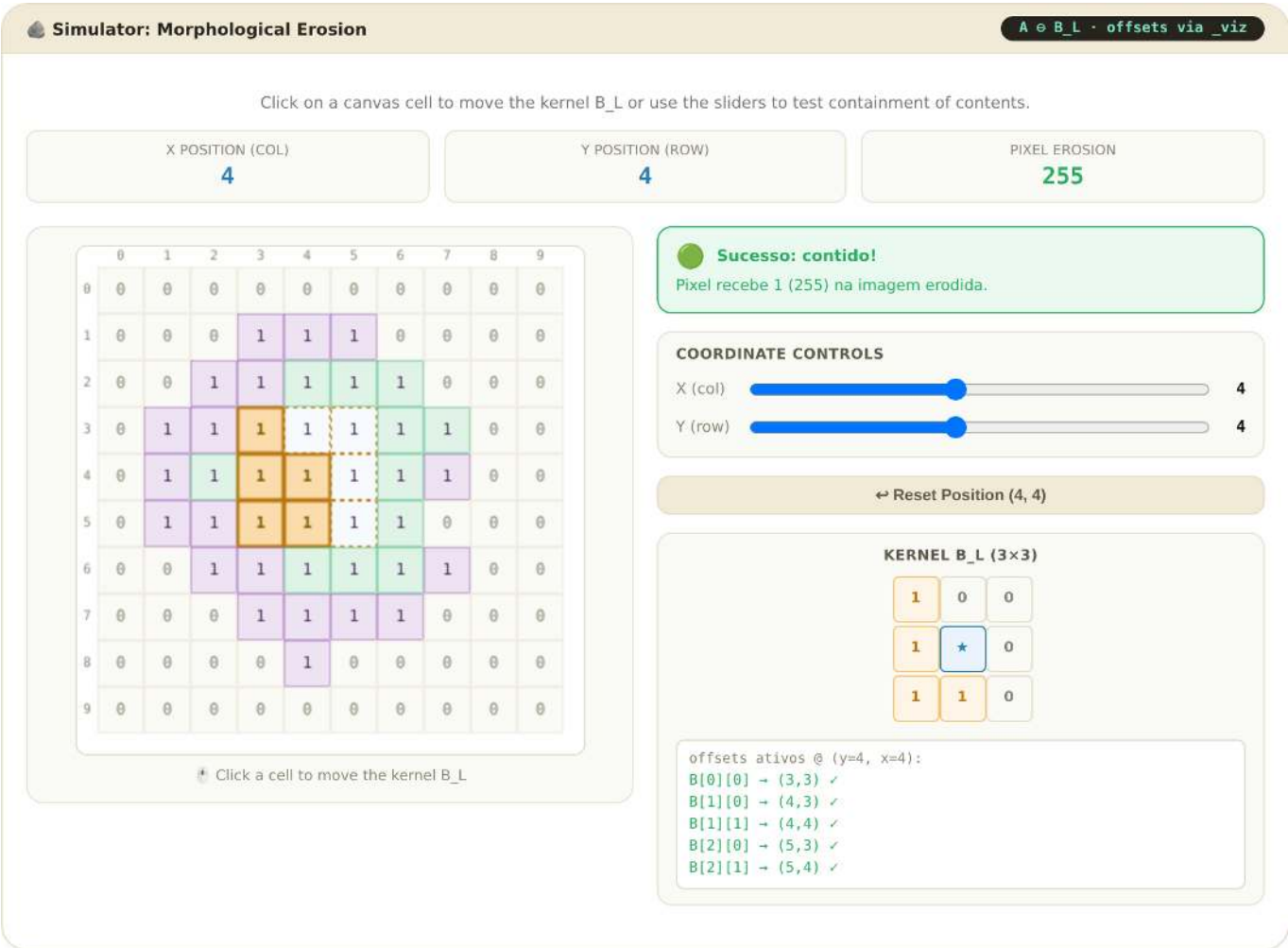


Figure 4.4: Erosion and dilation on a 10×10 binary image with an asymmetric 3×3 structuring element 'L'. Validation of the erosion–dilation duality.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-04-erosao.html

Figure 4.5: Simulator: Morphological Erosion (A B_L)

4.3.2 Opening and Closing

Combining erosion and dilation yields two operators of great practical utility: **opening** and **closing**, defined by Equations Equation 4.5 and Equation 4.6. Their main effects are summarized in Table 4.2.

Opening (*opening*) — erosion followed by dilation using the same B:

$$A \circ B = (A \ominus B) \oplus B \tag{4.5}$$

Closing (*closing*) — dilation followed by erosion using the same B:

$$A \bullet B = (A \oplus B) \ominus B \tag{4.6}$$

In practice, `mm.open` and `mm.close` apply the same structuring element in both stages. For symmetric structuring elements (the most common ones), this implementation coincides with the mathematical definition presented above.

Table 4.2: Properties of opening and closing.

Operator	Sequence	Main effect
Opening $A \circ B$	erosion $\rightarrow$ dilation	Removes structures unable to contain the structuring element; smooths external contours
Closing $A \bullet B$	dilation $\rightarrow$ erosion	Fills holes smaller than B; smooths internal contours

Important property: both are **idempotent**. For example,

$$(A \circ B) \circ B = A \circ B,$$

that is, after the first application, further applications of the same operator no longer alter the result.

4.3.2.1 Implementation of opening and closing

Unlike erosion and dilation, opening and closing do not introduce new computational mechanisms. Both are obtained by the sequential composition of the primitive operators already presented:

```
def open0(f, B):
    return mm.dil0(mm.ero0(f, B), B)

def close0(f, B):
    return mm.ero0(mm.dil0(f, B), B)
```

The `mm.open` function delegates the operation to `mm.open(f, B)`, while `mm.close` uses `mm.close(f, B)`, producing the same result more efficiently.

Opening inherits from erosion the ability to remove structures smaller than the structuring element and from dilation the partial restoration of preserved regions. Closing performs the reverse process: it first expands the objects and then restores their original dimensions, filling gaps and holes smaller than the structuring element.

4.3.2.2 Alternating Sequential Filter

In practice, opening and closing are often applied in sequence to simultaneously remove external noise and fill internal holes. The function `mm.asf` (*Alternating Sequential Filter*) generalizes this strategy by applying openings and closings alternately with progressively larger structuring elements. The available sequences are presented in Table 4.3.

Table 4.3: Sequences of the alternating sequential filter mm.asf.

Sequence	Order	Typical use
`'OC'`	opening → closing	removes external noise before filling small holes
`'CO'`	closing → opening	fills small holes before removing external noise
`'OCO'`	opening → closing → opening	emphasizes the removal of external noise
`'COC'`	closing → opening → closing	emphasizes the filling of holes and gaps

The parameter `n` controls the number of scales used by the filter. At each iteration i, the structuring element is enlarged by Minkowski addition (`mm.sesum(b, i)`), producing a sequence of increasingly comprehensive morphological filters. Unlike a single opening or closing with a large structuring element, the ASF performs progressive smoothing at multiple scales, better preserving the geometry of relevant objects while eliminating smaller structures. Figure 4.6 presents an example of applying opening, closing, and the ASF to the image of the coins.

```
img_bin    = mm.threshold(img_clahe)
B_disk     = mm.sedisk(19)

img_open  = mm.open(img_bin, B_disk)             # erosion → dilation
img_close = mm.close(img_bin, B_disk)            # dilation → erosion
img_oc    = mm.close(img_open, B_disk)           # opening followed by closing
img_asf   = mm.asf(img_bin, 'OC', mm.sedisk(3), n=7)
# ASF: base disk 3×3, grows each iteration

mm.show(
    [img_bin, img_open, img_close, img_oc, img_asf],
    titles=["Otsu binarization", "Opening (A B)", "Closing (A B)",
            "Opening→Closing", "ASF-OC (n=7)"],
    cols=5, figsize=(15, 12)
)
```

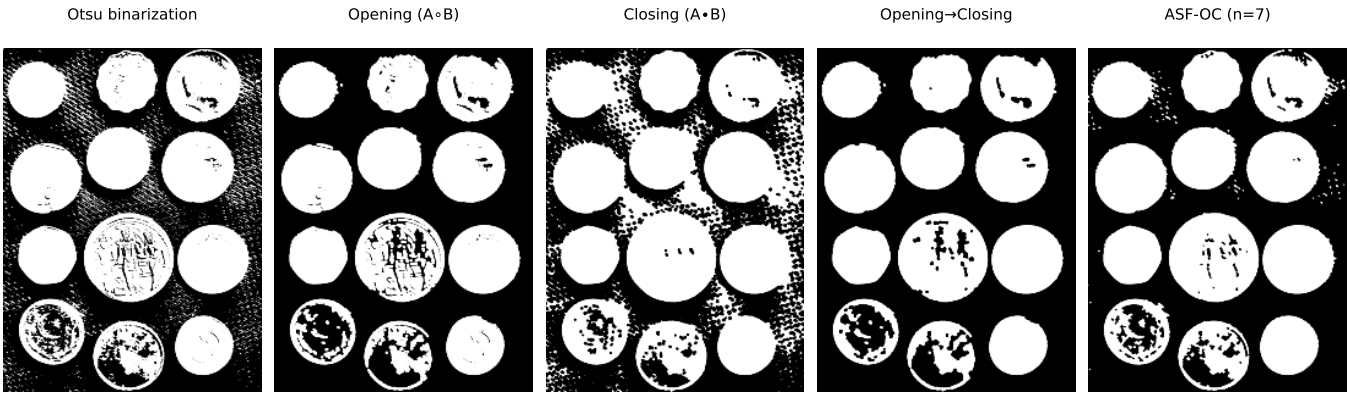


Figure 4.6: Opening, closing, composition, and alternating sequential filter applied to Otsu binarization of the coins (preprocessed by contrast enhancement). Structuring element: 13×13 disk.

4.3.3 Geodesic Operators

Geodesic operators introduce an additional constraint to classical morphological operators through a control image called the **mask** g. Instead of allowing erosion or dilation to propagate freely across the image, the result of each iteration is limited pointwise by the mask values, restricting the operation's evolution to permitted regions.

4.3.3.1 Geodesic Dilation

The **geodesic dilation** of a marker image f under a mask image g, using a flat structuring element b, is defined by:

$$f \oplus_g b = (f \oplus b) \wedge g, \tag{4.7}$$

where $\wedge$ represents the pointwise minimum.

In other words, a conventional dilation is initially performed on the marker, and then the result is constrained by the mask g. In this way, the propagation can never exceed the regions permitted by the mask.

The classical formulation of geodesic dilation assumes that the marker is contained within the mask, that is, $f \leq g$, ensuring that the evolution of the operation always remains bounded by the mask.

4.3.3.2 Implementation of geodesic dilation

The `mm.cdil` function directly implements this operator and allows executing multiple consecutive iterations:

```
def cdil(f, g, b=np.zeros((3,3),dtype='uint8'), n=1):
    """Dilatação geodésica do marcador f sob a máscara g."""
    y = f.copy()
    for _ in range(n):
        y = np.minimum(mm.dil(y, b), g)
    return y
```

The `np.minimum(mm.dil(y, b), g)` instruction exactly implements the mathematical definition of geodesic dilation, that is, $(y \oplus b) \wedge g$.

When $n = 1$, the function performs a single geodesic dilation. For $n > 1$, the result of each step becomes the marker for the next step, producing a progressive propagation controlled by the mask.

The interactive simulator Figure 4.7 allows you to follow, step by step, the propagation of the marker f along the corridors of the maze. At each iteration of `mm.cdil`, the dilation front advances to the neighboring free cells — those where $g = 1$ — while the walls ($g = 0$) remain impassable. The number of steps required for the marker to reach the exit corresponds exactly to the geodesic length of the shortest path within the mask, highlighting the direct connection between iterated geodesic dilation and the notion of distance in graphs.

4.3.3.3 Geodesic Erosion

Dually, the **geodesic erosion** of a marker image f under a mask image g is defined by:

$$f \ominus_g b = (f \ominus b) \vee g, \tag{4.8}$$

where $\vee$ represents the pointwise maximum operator.

In this case, the conventional erosion of the marker is followed by a lower constraint imposed by the mask. Thus, no pixel of the result can assume a value lower than the corresponding pixel of the mask.

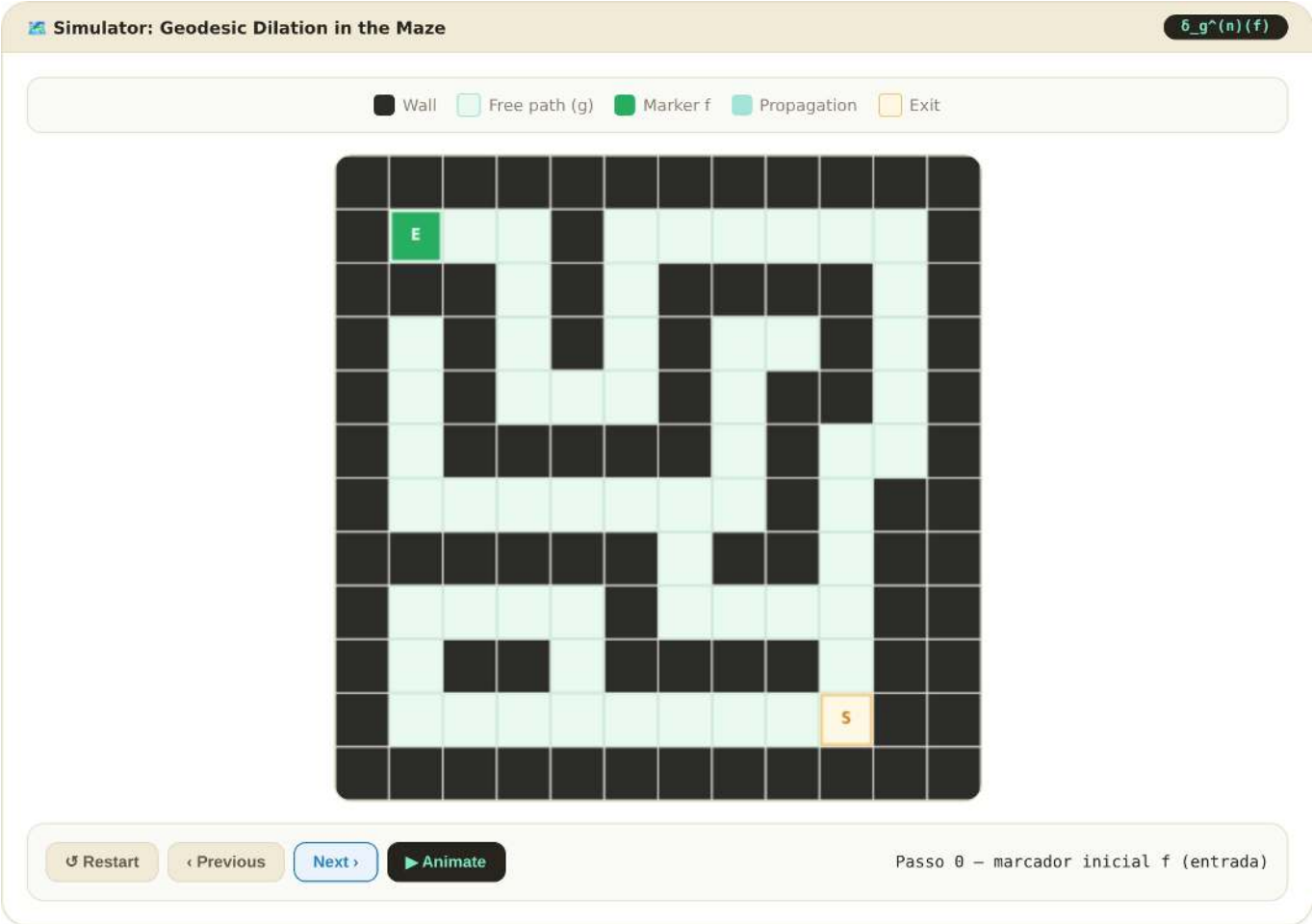
The classical formulation of geodesic erosion presupposes the dual condition

$$f \geq g,$$

so that the mask acts as a lower bound throughout the entire process.

4.3.3.4 Implementation of geodesic erosion

The static function `mm.cero` implements this operator:



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-04-cdil.html

Figure 4.7: Interactive simulator of geodesic dilation: the marker f (green) propagates step by step through the free paths of mask g, without crossing walls.

```
staticmethod
def cero(f, g, b=np.zeros((3,3),dtype='uint8'), n=1):
    """Erosão geodésica do marcador f sob a máscara g."""
    y = f.copy()
    for _ in range(n):
        y = np.maximum(mm.ero(y, b), g)
    return y
```

The instruction `np.maximum(mm.ero(y, b), g)` directly implements the expression $(y \ominus b) \vee g$.

As with geodesic dilation, the parameter n defines how many successive geodesic erosions will be computed before returning the final image.

Relation to morphological reconstruction

The morphological reconstruction presented in the next section is obtained by iteratively applying geodesic dilation (`mm.cdil`) until a fixed point is reached, that is, until no cell changes value between two consecutive iterations. In other words, reconstruction consists of a sequence of successive geodesic dilations that propagate within the mask until no further changes occur.
Dually, it is also possible to define reconstructions based on geodesic erosion through successive applications of `mm.cero`.

4.3.3.5 Example: propagation in a maze via duality

Figure 4.8 illustrates the resolution of the connectivity problem in a maze using morphological duality through the functions `mm.cero` and `mm.suprec`.

Instead of propagating a marker through the free corridors using geodesic dilations, the problem is formulated in the complementary domain. Initially, the original mask is inverted,

$$g = 1 - g_{orig},$$

so that the walls now assume value 1 and the corridors value 0. Similarly, the marker is constructed in this same complementary domain, containing a single value 0 at the maze entrance position and value 1 in all other pixels.

Using the cross-shaped structuring element (`mm.secross()`), the geodesic erosion acts on the complemented marker. At each iteration of `mm.cero`, the region connected to the initial marker undergoes successive erosions, while the mask imposes a lower limit that prevents propagation through the maze walls.

The intermediate images show evolution states after different numbers of iterations (`n=5`, `n=12`, and `n=22`). The final result is obtained by the geodesic reconstruction by erosion (`mm.suprec`), which applies successive geodesic erosions until reaching a fixed point, that is, a situation in which no further change occurs between two consecutive iterations.

In the complementary domain, the reconstructed region corresponds exactly to the set of corridors connected to the maze entrance. Thus, connectivity between entrance and exit can be determined directly from the reconstructed image.

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

# 1. Original mask g (0 = Wall, 1 = Corridor)
g_orig = np.array([
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 1, 1, 1, 1, 0, 1, 1, 1],
    [0, 0, 0, 0, 0, 1, 0, 1, 0, 1],
```

```
    [0, 1, 1, 1, 0, 1, 1, 1, 0, 1],
    [0, 1, 0, 1, 0, 0, 0, 0, 0, 1],
    [0, 1, 0, 1, 1, 1, 1, 1, 1, 1],
    [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
    [0, 1, 1, 1, 1, 1, 1, 0, 1, 0],
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]], dtype=np.uint8)

# Global inversion at the start (Morphological Duality)
g = 1 - g_orig  # Now: 1 = Wall, 0 = Corridor
f = np.ones((10, 10), dtype=np.uint8)
f[0, 1] = 0     # Seed injected as 0 in the corridor

# Cross structuring element
B_cruz = mm.secross()

# Direct processing using mm.cero and mm.suprec in the inverted domain
passo_5    = mm.cero(f, g, B_cruz, n=5)
passo_12   = mm.cero(f, g, B_cruz, n=12)
passo_22   = mm.cero(f, g, B_cruz, n=22)
ponto_fixo = mm.suprec(f, g, B_cruz)

# --- GRAPHICAL DISPLAY CONFIGURATION ---

titles = [
    "Máscara (~g)", "Marcador (~f)",
    "Avanço (n=5)", "Avanço (n=12)",
    "Avanço (n=22)", "~mm.suprec"
]
images = [g, f, passo_5, passo_12, passo_22, ponto_fixo]

fig, axes = plt.subplots(1, 6, figsize=(16, 4), facecolor='#fcfcfc')

# Color map adapted for the complementary domain:
# In the inverted domain: 1 = Wall (Dark Blue)
# Where g == 0 and image == 1 = Free Corridor (Light Gray)
# Where g == 0 and image == 0 = Active Geodesic Wave (Gold)
cmap_pipeline = mcolors.ListedColormap(['#ffc13b', '#e0e0e0', '#1e3d59'])

for i, ax in enumerate(axes):
    if i == 0 or i == 1:
        # For the initial conditions (~g and ~f)
        cmap_init = mcolors.ListedColormap(['#e0e0e0', '#1e3d59'])
        ax.imshow(images[i], cmap=cmap_init, vmin=0, vmax=1)
    else:
        # Rendering based on complementary states
        render_step = np.zeros_like(g, dtype=np.uint8)
        render_step[g == 1] = 2           # Wall (1 original from the color map)
        render_step[g == 0] = 1           # Default corridor
        render_step[images[i] == 0] = 0   # Active wave (0 original from the color map)
        ax.imshow(render_step, cmap=cmap_pipeline, vmin=0, vmax=2)

    ax.set_title(titles[i], fontsize=10, fontweight='bold', color='#1e3d59', pad=12)
    ax.set_xticks(np.arange(-0.5, 10, 1), minor=True)
    ax.set_yticks(np.arange(-0.5, 10, 1), minor=True)
    ax.grid(which='minor', color='#ffffff', linestyle='-', linewidth=1)
    ax.tick_params(which='both', bottom=False, left=False, labelbottom=False, labelleft=False)

    # Graphical indicators for Entry and Exit
    ax.plot(1, 0, marker='v', color='#2ecc71', markersize=7, markeredgewidth=1.5)
```

```
    ax.plot(8, 9, marker='o', color='#e74c3c', markersize=6, fillstyle='none',
markeredgewidth=2)

plt.tight_layout()
plt.show()
```

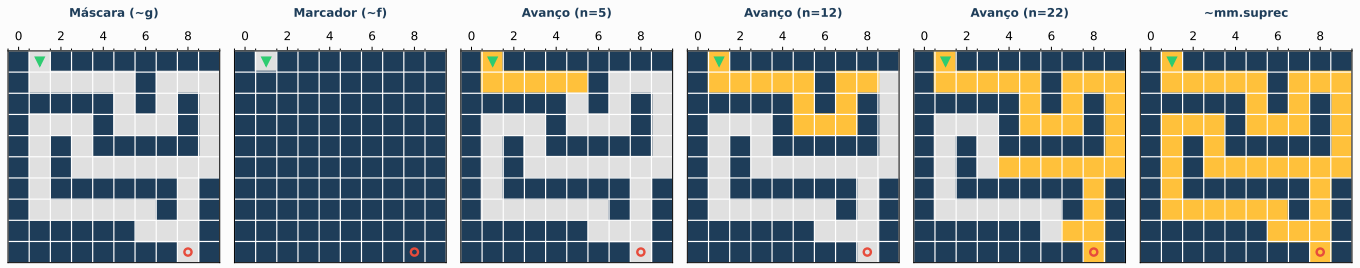


Figure 4.8: Maze resolution via Complementary Operations: Inverting the mask and marker at the start of the *pipeline*, the flow is resolved directly in the complementary domain through mm.cero and mm.suprec, eliminating redundant re-inversions.

4.3.4 Morphological Reconstruction

Morphological reconstruction propagates a **marker** image f within a **mask** image g, ensuring that the result never exceeds the intensity values imposed by the mask. The fundamental operator that enables this contained propagation is the **geodesic dilation**, defined by Equation 4.7.

Reconstruction is obtained by the iterative application of this conditioned dilation. Initially, the marker is bounded by the mask to establish the initial state:

$$X^{(0)} = f \wedge g,$$

and the subsequent iterations are defined recursively by:

$$X^{(k)} = (X^{(k-1)} \oplus b) \wedge g.$$

The sequence grows monotonically until it reaches a fixed point, producing the **morphological reconstruction by dilation** (also known in the literature as *inf-reconstruction*):

$$R_g^{\delta}(f) = \lim_{k\to\infty} X^{(k)} = X^{(k)} \quad \text{when} \quad X^{(k)} = X^{(k-1)}. \tag{4.9}$$

The iterative ascent is halted as soon as stability is achieved, that is, when two consecutive iterations produce matrices with absolutely identical values.

4.3.4.1 Implementation of morphological reconstruction

The didactic routine `mm.infrec` directly implements the iterative fixed-point algorithm. Initially, the initial effective marker $X^{(0)}$ is determined by the operation `np.minimum(f, g)`. To ensure that the verification loop executes the first pass without triggering false premature convergences, the control variable of the previous iteration (`y1`) is initialized filled with a sentinel value outside the data domain (or simply with a matrix that forces the first execution).

```
def infrec(f, g, b=np.zeros((3,3), dtype='uint8')):
    """Inf-reconstruction: dilates the marker (f   g) until convergence under the mask g."""
    y = np.minimum(f, g)
    # Initialize y1 with impossible values to force entry into the loop
    y1 = np.full_like(f, 256, dtype=np.int16)
```

```
    while not np.array_equal(y, y1):
        y1 = y.copy()
        # Apply geodesic dilation: (y   b)   g
        y = np.minimum(mm.dil(y, b), g)
    return y.astype('uint8')
```

Inside the `while` loop, the variable `y` stores the current estimate of the reconstruction $X^{(k)}$, while `y1` preserves the image from the immediately preceding stage $X^{(k-1)}$. The conditional control statement `np.minimum(mm.dil(y, b), g)` faithfully translates the theoretical geodesic dilation, where the conventional morphological expansion commanded by OpenCV is immediately "pruned" and limited by the intensity barriers of the mask g. The loop ceases when no pixel modification is recorded between steps.

4.3.4.2 Advantages of Morphological Reconstruction

Morphological reconstruction is significantly more robust than conventional opening because it can remove unwanted structures without distorting or altering the morphology of the objects that should be preserved.

While classical opening smooths corners, eliminates tips, and deforms contours due to the rigid geometric imposition of the structuring element, geodesic reconstruction uses the mask to accurately recover the original boundaries and shapes of objects that have connectivity with the original marker.

In intuitive terms, the marker acts as a contagion seed that expands progressively, but only travels through the regions allowed by the mask. Components that have no intersection with the marker will never be reconstructed (being eliminated), while components touched by the seed expand until fully restoring their original geometry.

This discriminatory and conservative behavior is illustrated in Figure 4.9, using the cross-shaped structuring element presented in Figure 4.3.

```
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()
f = mm.ero(g, mm.sebox())          # marker: nucleus of the main object

iteracoes, titulos = [], []
img_atual = f
for i in range(1, 6):
    img_atual = mm.cdil(img_atual, g, B_cruz)
    iteracoes.append(img_atual)
    titulos.append(f"Dilatação Cond. (n={i})")

img_reconstruida = mm.infrec(f, g, B_cruz)
print(f"  Stability at iteration 5: {np.array_equal(img_reconstruida, iteracoes[-1])}")

mm.show(
    [g, f] + iteracoes + [img_reconstruida],
    titles=["Máscara (g)", "Marcador (f)"] + titulos + ["Reconstrução R_g(f)"],
    cols=8, figsize=(18, 3)
)
```

```
Stability at iteration 5: True
```

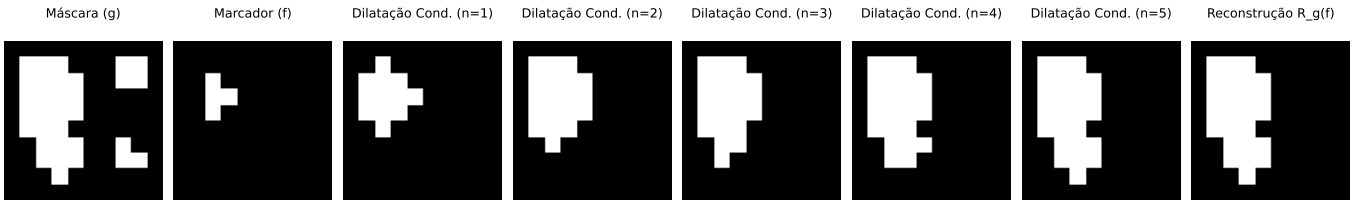


Figure 4.9: *Morphological Reconstruction Pipeline by Conditioned Dilation: the mask contains two objects, the marker isolates only the nucleus of the main object, and the iterations rebuild its exact shape until convergence.

4.3.5 Hole Filling and Border Removal

Two operators based on morphological reconstruction complete the binary cleaning *pipeline*. Their main characteristics are summarized in Table 4.4.

Hole filling (`mm.clohole`) removes cavities completely surrounded by the object, regardless of size, without altering the external contours. The procedure operates on the complement of the image, using as a marker a restriction of the frame to the background:

$$\text{clohole}(f) = \left(R^{\delta}_{f^c}(\text{frame}(f) \wedge f^c)\right)^c \tag{4.10}$$

In operational terms, the external background is first reconstructed, and then complementation is applied to recover the objects with filled holes.

Border object removal (`mm.edgeoff`) eliminates all objects that touch the image border, preserving only fully internal components. The marker is obtained by the intersection between the frame and the objects of the image:

$$\text{edgeoff}(f) = f \setminus R^{\delta}_{f}(\text{frame}(f) \wedge f) \tag{4.11}$$

Table 4.4: Comparison between the *clohole* and *edgeoff* operators.

Operator	Marker	Mask	Effect
`mm.clohole`	*frame* restricted to the background (f^c)	f^c	Fills internal holes
`mm.edgeoff`	*frame* restricted to the object (f)	f	Removes objects connected to the border

The step-by-step evolution of these geodesic transformations can be followed in the figures below. Figure 4.10 illustrates the controlled flooding mechanism of the `mm.clohole` operator, in which reconstruction occurs from the external background and prevents propagation into internal regions not connected to the exterior, resulting in consistent filling of internal cavities. In contrast, Figure 4.11 details the dynamics of the `mm.edgeoff` operator, in which only components connected to the border are reconstructed and subsequently removed, preserving exclusively the objects fully contained within the interior of the image.

The connectivity of the geodesic propagation is controlled by the structuring element: `mm.sebox()` (8-neighborhood) includes diagonal connections, whereas `mm.secross()` (4-neighborhood) excludes them. Consequently, the choice of the structuring element affects which components are reached by the reconstruction and, therefore, which will be preserved or removed.

4.3.5.1 Compliance with the implementation

The definitions above are directly aligned with the implementation in `morph.py`, reproduced below:

```
staticmethod
def clohole(f, b=np.ones((3,3),dtype='uint8')):
    # marcador restrito ao fundo da imagem
    marcador = mm.frame(f, border=1) & mm.neg(f)
    return mm.neg(mm.infrec(marcador, mm.neg(f), b))

staticmethod
def edgeoff(f, b=np.ones((3,3),dtype='uint8')):
    # marcador restrito aos objetos da imagem
    marcador = mm.frame(f, border=1) & f
    return mm.subm(f, mm.infrec(marcador, f, b))
```

These implementations make it explicit that both operators are direct instances of morphological reconstruction by geodesic dilation using `mm.infrec`, differing only in the choice of marker and mask: `clohole` operates on the image complement, while `edgeoff` operates directly in the domain of objects.

```
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)
marcador_ch = mm.frame(f, border=1)       # external border

iteracoes, titulos = [], []
img_atual = marcador_ch
for i in range(1, 6):
    img_atual = mm.cdil(img_atual, f_c, B_cruz)
    iteracoes.append(img_atual)
    titulos.append(f"Iter. (n={i})")

img_clohole = mm.neg(mm.infrec(marcador_ch, f_c, B_cruz))
print(f"  Clohole validation: {np.array_equal(img_clohole, mm.clohole(f))}")

mm.show(
    [f, marcador_ch] + iteracoes + [img_clohole],
    titles=["f original", "Marcador (borda)"] + titulos + ["clohole(f)"],
    cols=8, figsize=(18, 3), axis=True
)
```

```
  Clohole validation: True
```

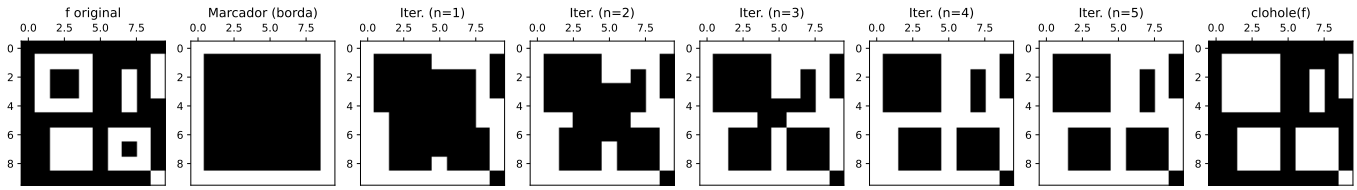


Figure 4.10: *Hole-filling pipeline* (*clohole*): the marker comes from the image border, restricted to the complement f^c. Geodesic dilation reconstructs the external background; after complementation, the internal holes are filled.

```
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_box = mm.sebox()
marcador_eo = mm.band(mm.frame(f, border=1), f)   # border   f

iteracoes, titulos = [], []
img_atual = marcador_eo
for i in range(1, 6):
    img_atual = mm.cdil(img_atual, f, B_box)
    iteracoes.append(img_atual)
    titulos.append(f"Iter. (n={i})")

img_edgeoff = mm.edgeoff(f, B_box)

mm.show(
    [f, marcador_eo] + iteracoes + [img_edgeoff],
    titles=["f original", "Marcador (borda)"] + titulos + ["edgeoff(f)"],
    cols=8, figsize=(18, 3)
)
```

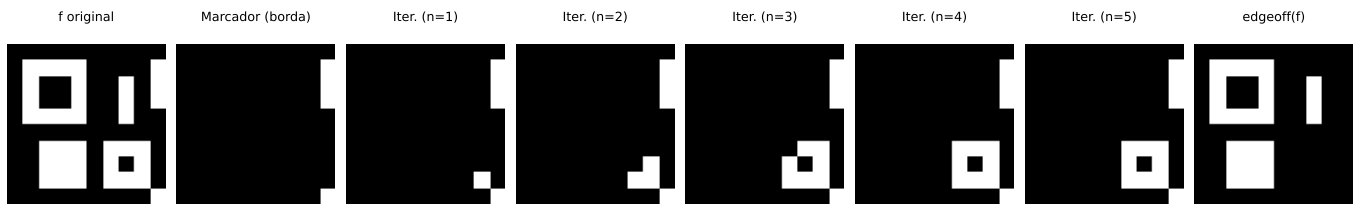


Figure 4.11: *Pipeline* for removing border structures (*edgeoff*): the marker captures roots connected to the extremities, reconstruction delimits these elements, and subtraction preserves only fully internal objects.

4.3.6 Binary Cleaning Pipeline with CLAHE

Based on the previous analysis — in which CLAHE produced the highest inter-class variance value ($\sigma_B^2 \approx 2.47 \times 10^3$), see Figure 4.2, and the `mm.clohole` operator proved effective in filling internal cavities —, the final segmentation pipeline, illustrated in Figure 4.12, is structured by the following computational flow:

$$\text{gray} \xrightarrow{\text{CLAHE}} \xrightarrow{\text{Otsu}} \xrightarrow{\text{open}} \xrightarrow{\text{clohole}} \xrightarrow{\text{open}} \xrightarrow{\text{edgeoff}} \text{segmentation}$$

After the `mm.clohole` step, a second morphological opening is applied with a larger structuring element (`mm.sedisk(33)`, a disk of diameter 33). This operation removes small residual regions and artifacts that may have remained after segmentation. In particular, geodesic filling can transform small isolated cavities into components connected to the object, making an additional size-based filtering step convenient. The diameter was chosen so that the coins remain able to contain the structuring element, while significantly smaller components are eliminated.

In this image, no coin is connected to the matrix border. Consequently, applying `mm.edgeoff` does not alter the result obtained after the second opening. Nevertheless, this step is kept in the pipeline for robustness, since in other images there may be partially visible objects or objects connected to the borders, which should be removed before the analysis stage.

Why the opening after clohole?

The `mm.clohole` operator fills all closed cavities present in the segmented objects. In some situations, small unwanted regions may remain after this step or become connected to the main objects. The subsequent morphological opening removes components smaller than the structuring element, while preserving the coins due to their significantly larger size.

```
#   Preprocessing: CLAHE only
img_clahe0 = mm.clahe(img_coins_gray, 2.0, 8)

#   Step 1: Otsu binarization
img_bin = mm.threshold(img_clahe0)

#   Step 2: Opening - removes white background noise
img_open = mm.open(img_bin, mm.sedisk(9))

#   Step 3: clohole - closes all internal holes
img_hole = mm.clohole(img_open)

#   Step 4: Opening (large kernel) - removes clohole artifacts
img_limpo = mm.open(img_hole, mm.sedisk(33))

#   Step 5: edgeoff - removes objects touching the border
img_final = mm.edgeoff(img_limpo, border=1)

mm.show(
    [img_clahe0,  img_bin,              img_open,
     img_hole,    img_limpo,            img_final],
    titles=["CLAHE",            "Otsu",                "Opening (r=9)",
            "clohole",          "Opening (r=33)",     "Final (edgeoff)"],
    cols=6, figsize=(18, 6)
)
```

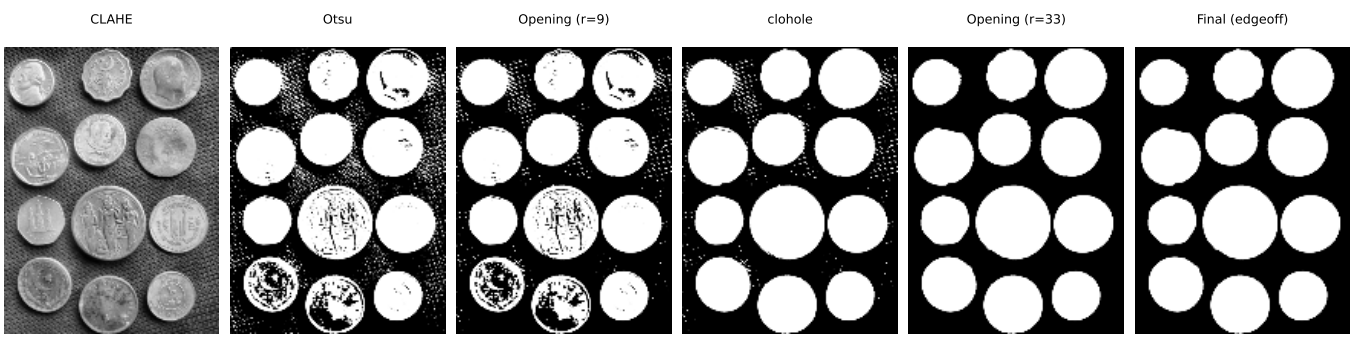


Figure 4.12: Complete *pipeline* of segmentation with CLAHE: Otsu → opening (r=9) → clohole → opening (r=33) → edgeoff.

4.3.7 Grayscale Morphology

Morphological operators extend naturally to grayscale images. In this formulation, erosion and dilation act directly on the image intensity levels. For flat structuring elements ($b \equiv 0$), erosion corresponds to the **local minimum** and dilation to the **local maximum** within the neighborhood defined by the structuring element.

The intuitive interpretation is straightforward: erosion **darkens** regions by replacing each pixel with the smallest value in its neighborhood, while dilation **lightens** regions by using the largest available value. Combining these operators allows for the construction of transformations capable of enhancing edges, removing illumination trends, and highlighting local structures.

Three derived operators are especially useful:

Morphological gradient — highlights edges as the difference between dilation and erosion:

$$\text{grad}_B(f) = (f \oplus B) - (f \ominus B) \tag{4.12}$$

Top-hat — highlights bright structures smaller than the structuring element (difference between the original image and its opening):

$$\text{top-hat}_B(f) = f - (f \circ B) \tag{4.13}$$

Black-hat — highlights dark structures smaller than the structuring element (difference between the closing and the original image):

$$\text{black-hat}_B(f) = (f \bullet B) - f \tag{4.14}$$

The *Top-hat* extracts bright details that do not survive the opening, while the *Black-hat* reveals dark details removed by the closing. The morphological gradient, in turn, enhances abrupt intensity transitions, producing a representation similar to that of an edge detector.

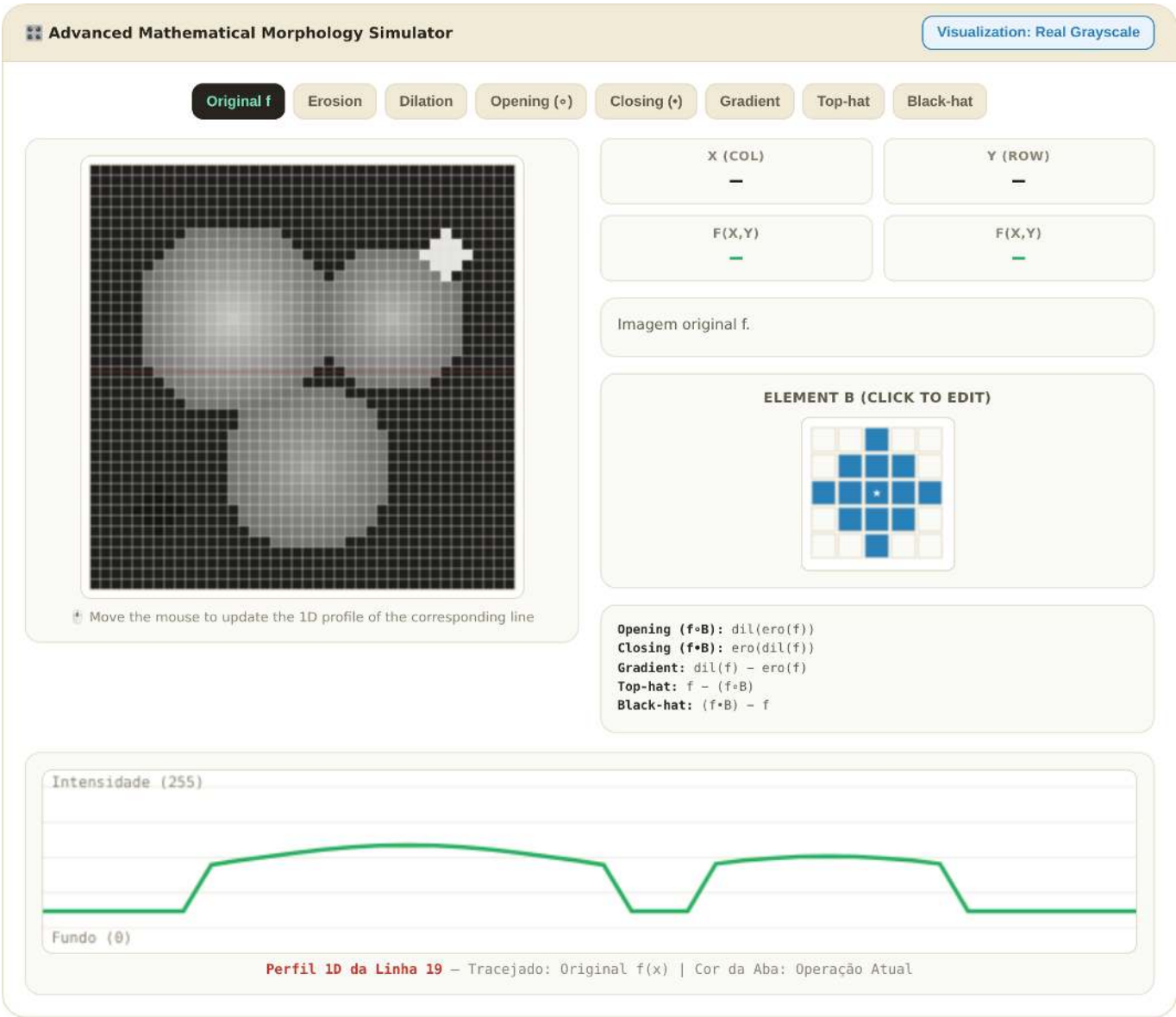
To understand the mechanism of these operators at a local level, Figure 4.13 presents an interactive simulator of grayscale morphology. The simulator allows you to freely edit the structuring element, visualize its displacement over the image, and simultaneously track the one-dimensional intensity profile. In this way, it becomes possible to directly observe how erosion selects local minima, how dilation selects local maxima, and how the morphological gradient emerges from the difference between these two operators.

The button located in the upper right corner allows you to toggle between the original grayscale visualization and a pseudocolored representation (*colormap*) for the three gradient types only. The colored version facilitates the visual perception of intensity variations, making the action of the morphological operators on maxima, minima, and local transitions of the image more evident.

The operators presented are available in `morph.py` through the functions `mm.gradm`, `mm.tophat`, and `mm.blackhat`.

Figure 4.14 illustrates the effects of these operators on the coin image and their respective histograms. Note that erosion shifts the distribution toward lower intensities, while dilation shifts it toward higher intensities. The gradient concentrates values in contour regions, and the *Top-hat* and *Black-hat* operators produce histograms strongly concentrated at low intensity levels, as only small local structures are highlighted.

```
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)
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-04-operadores-av.html

Figure 4.13: Advanced interactive morphology simulator with editable structuring element and 1D profile.

```python
def plot_hist(img, title):
    fig, ax = plt.subplots(figsize=(4, 3))
    h = mm.hist(img)
    # CORRECTION: range dynamically uses len(h) to match the library's return
    ax.bar(range(len(h)), h, color='steelblue', width=1, edgecolor='steelblue')
    ax.set(xlim=(0, 255)); plt.tight_layout()
    return fig2img(fig)

# 1. Basic morphological processing
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. Dynamic assembly of the pair: [Image, Histogram]
imgs, titles = [], []
for nome, img in operadores:
    imgs += [img, plot_hist(img, f"Hist. {nome}")]
    titles += [nome, f"Hist. {nome}"]

# 3. Final display in a two-column grid (Image | Histogram)
mm.show(imgs, titles=titles, cols=4, figsize=(10, 8))
```

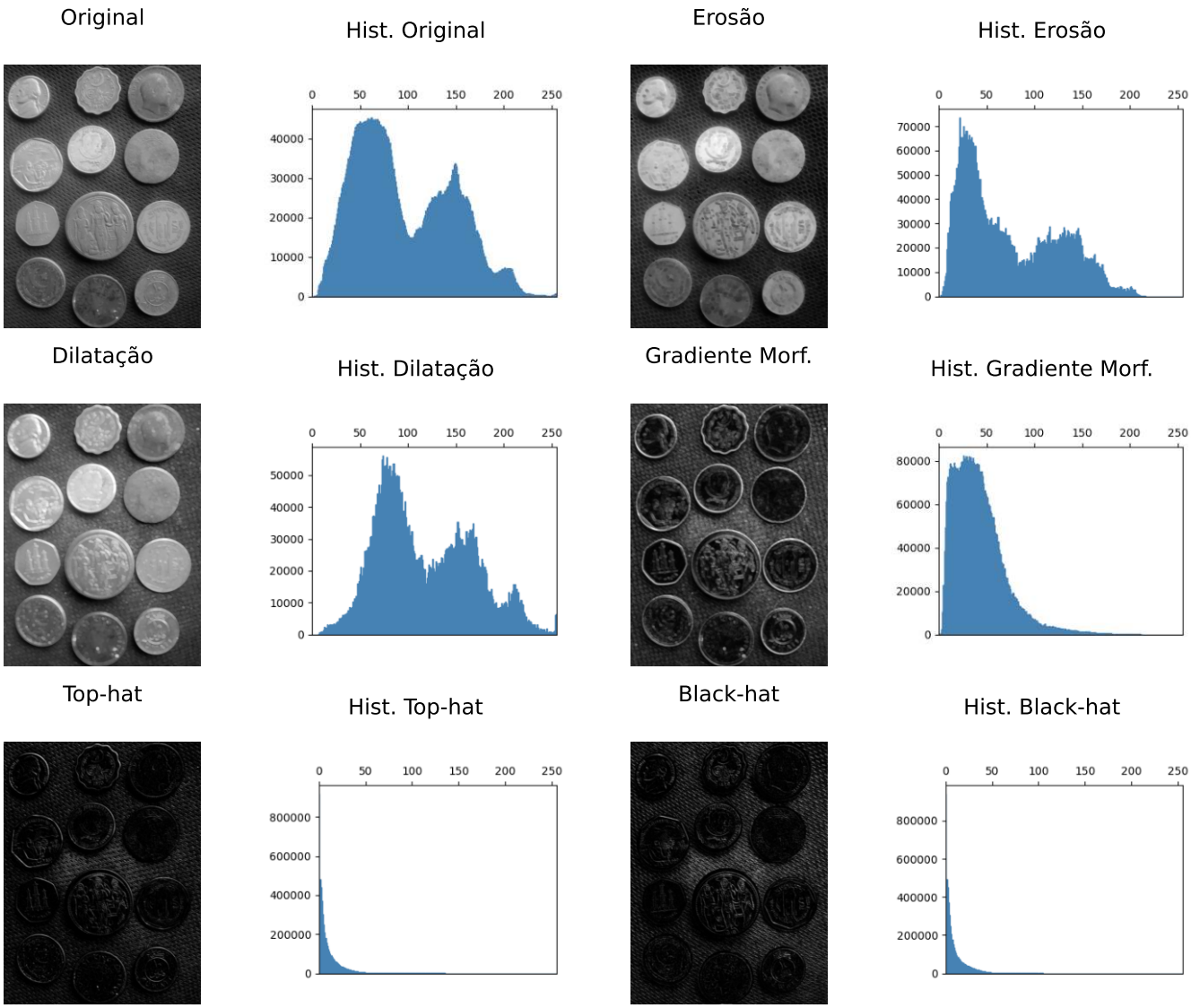


Figure 4.14: Grayscale morphology and their respective histograms. Erosion reduces local intensities, dilation amplifies them, the gradient highlights edges, and the Top-hat and Black-hat operators highlight bright and dark local details. Structuring element: disk of diameter 19.

The morphological operators presented previously will now be used as tools for refinement and marker generation in more advanced segmentation methods, presented below.

4.4 Image Segmentation: Fundamentals and Taxonomy

Image segmentation consists of dividing the image into regions associated with objects or structures of interest. In DIP, it represents the transition between low-level processing — such as filtering and enhancement — and more advanced analysis stages, such as feature extraction, recognition, and scene interpretation.

Formally, the goal of segmentation is to decompose the complete spatial domain of an image, denoted by Ω, into a partition of subsets $\{R_1, R_2, \dots, R_n\}$ that simultaneously satisfies the criteria of **completeness** and **disjointness**:

$$\bigcup_{i=1}^{n} R_i = \Omega, \qquad R_i \cap R_j = \emptyset \quad \forall\, i \neq j \tag{4.15}$$

In addition to the completeness and disjointness properties expressed in Equation 4.15, each subregion R_i

must constitute a **homogeneous** domain according to a similarity predicate defined over local properties — intensity, color, or texture — and, simultaneously, be **distinct** from adjacent regions.

Segmentation techniques can be organized into different families. This chapter emphasizes the approaches summarized in Table 4.5, based primarily on criteria of intensity, connectivity, and spatial proximity.

Table 4.5: Simplified taxonomy of the main segmentation and refinement approaches studied in this chapter.

Approach	Segmentation Criterion	Reference Operators
Thresholding	Partitioning of the intensity space	Otsu's criterion, global and local thresholding
Mathematical Morphology	Spatial relations defined by structuring elements/functions	Erosion, dilation, opening, closing, and reconstruction
Region-Based	Local homogeneity and spatial connectivity	Connected component labeling, Distance Transform, and *Watershed*

Up to this point, the practical development has focused on **thresholding**, through the combination of adaptive CLAHE equalization and Otsu's global method. This step was complemented by **morphological reconstruction** operators based on geodesic dilations, implemented by the functions `mm.infrec`, `mm.clohole`, and `mm.edgeoff`, producing a clean binary mask suitable for analysis.

However, in scenarios where distinct objects appear connected in the binary mask — whether by physical contact, partial overlap, or narrow pixel bridges produced by segmentation — thresholding is no longer sufficient to individualize each object. In such cases, multiple objects become part of a single connected component, hindering subsequent measurement and interpretation stages.

To overcome this limitation, the following sections introduce three complementary tools: **Connected Component Labeling**, the **Distance Transform**, and the marker-based ***Watershed*** segmentation algorithm. Together, these techniques make it possible to separate adjacent objects, identify regions individually, and extract consistent geometric descriptors for quantitative analysis.

4.4.1 Labeling

Connected component labeling is the operator that assigns a unique integer identifier to each set of pixels belonging to the same connected component in a binary image.

> **Formal Definition**
>
> Given a binary image f and a connectivity relation defined by a structuring element B (typically 4-connectivity or 8-connectivity), the labeling algorithm of Figure 4.15 produces an image g in which all pixels belonging to the same connected component receive the same positive integer label, while pixels belonging to distinct components receive different labels.

Connectivity defines which pixels are considered direct neighbors of a pixel (x, y). The most commonly used definitions are:

- **4-Connectivity:** considers only the four orthogonal neighbors (north, south, east, and west).
- **8-Connectivity:** considers the four orthogonal neighbors and the four diagonal ones, totaling eight neighbors.

The choice of connectivity directly influences the formation of connected components and, consequently, the labeling result, as illustrated in Figure 4.17. An additional example can be interactively explored in the simulator presented in Figure 4.16.

The implementation in `morph.py` provides two versions of this operator. The function `mm.label0` explicitly reproduces the flood-fill algorithm using a stack and allows controlling connectivity through the adopted

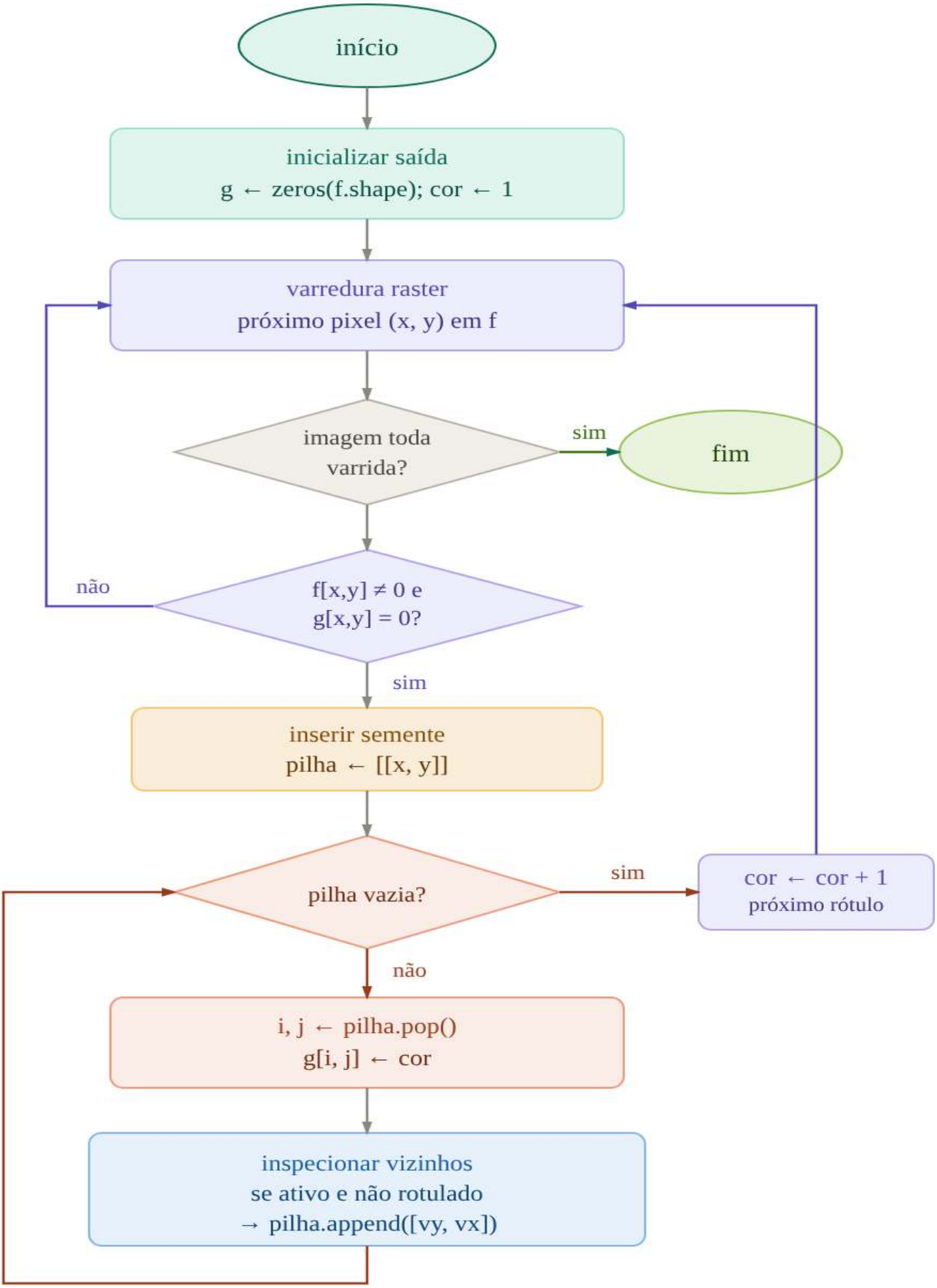
structuring element. Meanwhile, `mm.label` delegates the operation to OpenCV's optimized implementation (`mm.label0`). In both cases, the result is a labeled image in which each connected component receives a distinct integer identifier.

The helper `_viz` iterates over the structuring window `b` centered at (i, j), generating only the valid neighbors within the image boundaries — the desired connectivity is entirely determined by the shape of `b` passed to the algorithm.

Didactic example — effect of connectivity:

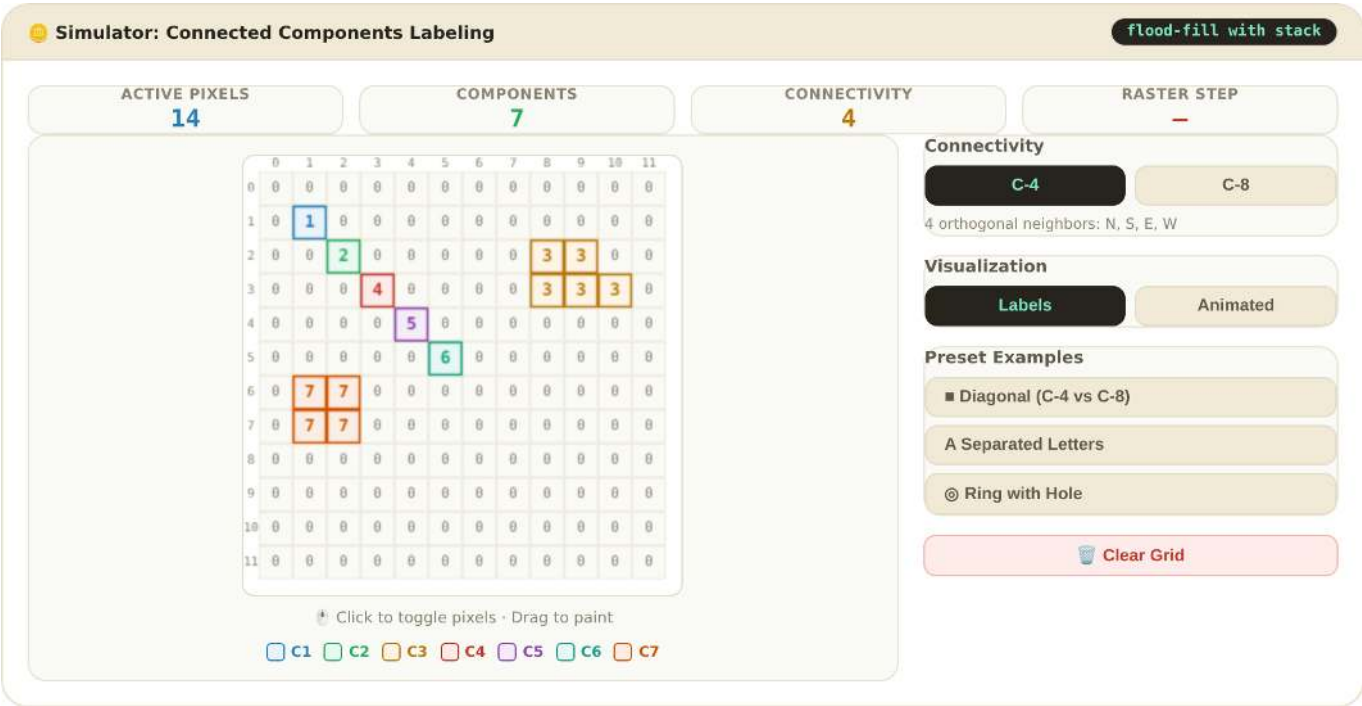
```
# 10×10 binary image with diagonally adjacent components
f = np.array([
    [0,0,0,0,0,0,0,0,0,0],
    [0,1,0,0,0,0,0,0,0,0],
    [0,0,1,0,0,0,0,0,0,0],  # diagonal with previous line
    [0,0,0,1,0,0,1,1,0,0],
    [0,0,0,0,0,0,1,1,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,1,1,1,0,0,0,0,0,0],
    [0,1,0,1,0,0,0,1,0,0],
    [0,1,1,1,0,0,0,0,1,0],  # diagonal with previous line
    [0,0,0,0,0,0,0,0,0,0]], dtype=np.uint8) * 255

# Cross structuring element (4-connectivity) and square (8-connectivity)
B4 = mm.secross()   # 4-connectivity
B8 = mm.sebox()     # 8-connectivity

# Labeling with label0 (didactic) and 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)

# Cross-validation
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"   Components C4: {lbl4_cv2.max()}  |  Components C8: {lbl8_cv2.max()}")

# Normalization for visualization
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"C4 labeling\n({lbl4_cv2.max()} components)",
        f"C8 labeling\n({lbl8_cv2.max()} components)"
    ],
    cols=3, figsize=(12, 4), axis=True
)
```

```
  label0 (C4) == cv2 (C4): True
  label0 (C8) == cv2 (C8): True
   Components C4: 7  |  Components C8: 4
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-alg-rotulagem2.html

Figure 4.15: Flood-fill labeling algorithm with stack.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-04-rotulacao.html

Figure 4.16: Interactive simulator for connected component labeling: visualization of flood-fill expansion, 4-connectivity and 8-connectivity.

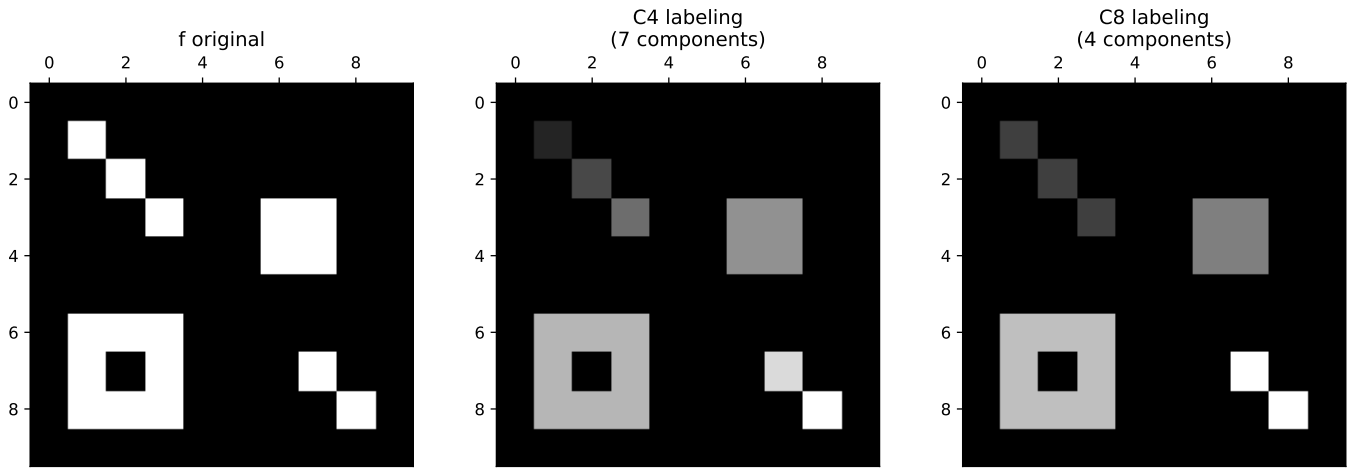


Figure 4.17: Effect of connectivity on labeling: the 10×10 binary image contains diagonally adjacent pixels. With 4-connectivity, these pixels form distinct components; with 8-connectivity, they merge into a single component.

4.4.2 Distance Transform

The **Distance Transform** (DT) is an operator that, applied to a binary image f, produces a grayscale image D in which each pixel belonging to the object ($f(x, y) \neq 0$) receives as its value the geometric distance to the nearest background pixel ($f(x', y') = 0$):

$$D(x, y) = \min_{(x',y') \,:\, f(x',y')=0} d\big((x, y),\, (x', y')\big)$$

where $d(\cdot, \cdot)$ is a distance metric — typically the Euclidean distance (L_2). The result is a topographic representation of the objects: pixels located in the interior assume high values, whereas pixels near the

borders exhibit low distance values. The **local maxima** of D correspond to the points farthest from the object boundary, often near their geometric centers or centers of maximal inscription — a property particularly useful for the automatic generation of markers in the *watershed* algorithm.

> **Formal definition using erosions**
>
> The DT also admits an iterative morphological interpretation, according to the algorithm in Figure 4.18. Consider a structuring function b whose central value is zero and whose neighbors have negative costs associated with the displacement. By applying successive erosions with this particular structuring function, the pixel values of the objects (which must assume the maximum possible distance in the image) are progressively reduced according to the costs defined by b. The accumulated value of this propagation then comes to represent the distance to the background according to the metric induced by the structuring function.

This interpretation is implemented in `mm.dist1()`, which accumulates successive erosions using the operation `mm.ero1()`. In turn, `mm.dist()` delegates the Euclidean distance computation to the optimized OpenCV operator `mm.dist(f)`, where `f` is the input binary image, the L2 distance specifies the Euclidean metric (L_2), and `5` indicates the use of a 5×5 mask to approximate the distance with high precision.

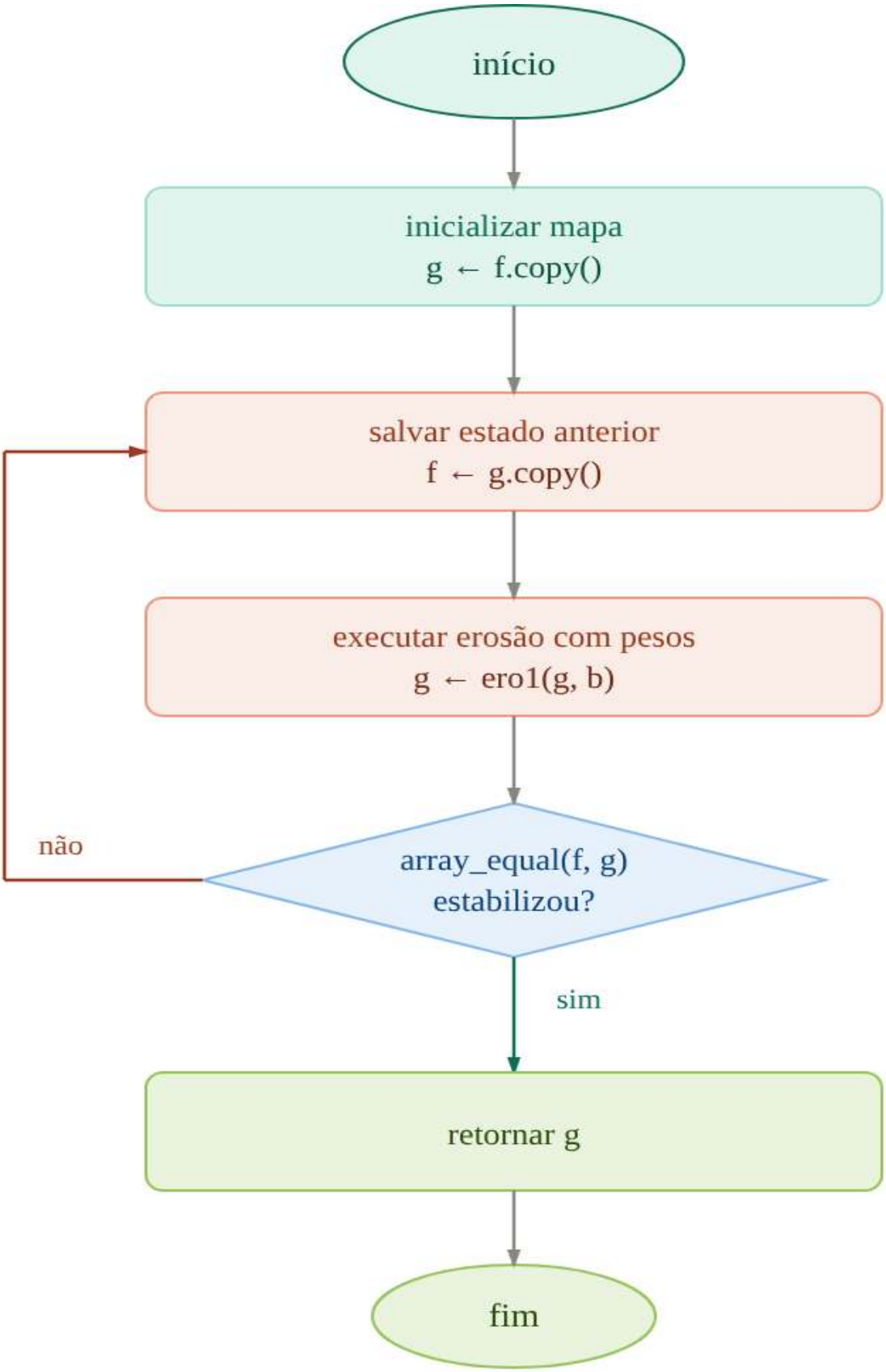
The `dist1` function produces a discrete distance transform whose metric is determined by the geometry and weights of the structuring function used. For example, using a cross-shaped structuring function with unit cost for the four orthogonal neighbors yields the ***Manhattan*** distance (L_1). Other choices of neighborhood and weights induce different metrics. In turn, `mm.dist()` computes an efficient approximation of the Euclidean distance (L_2).

Because it requires successive erosions over the entire image, the `dist1` approach has a significantly higher computational cost than `mm.dist()`, and it is employed in this book mainly for didactic purposes and to highlight the relationship between mathematical morphology and distance transforms.

Figure 4.19 presents an interactive simulator for the TD: by positioning the cursor over different pixels of the object, it is possible to observe, in real time, the distance value associated with that position—that is, the distance to the nearest background pixel. Figure 4.20 presents a practical example of this execution in a Python environment.

```
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"Max dist1 (erosions): {int(d_iter.max())} px")
print(f"Max dist (L2)      : {int(d_l2.max())} px")

mm.show(
    [f, d_iter, d_l2],
    titles=["original f", "mm.dist1 (erosions with cross)", "mm.dist (L2)"],
    cols=3, figsize=(12, 4), axis=True
)
```

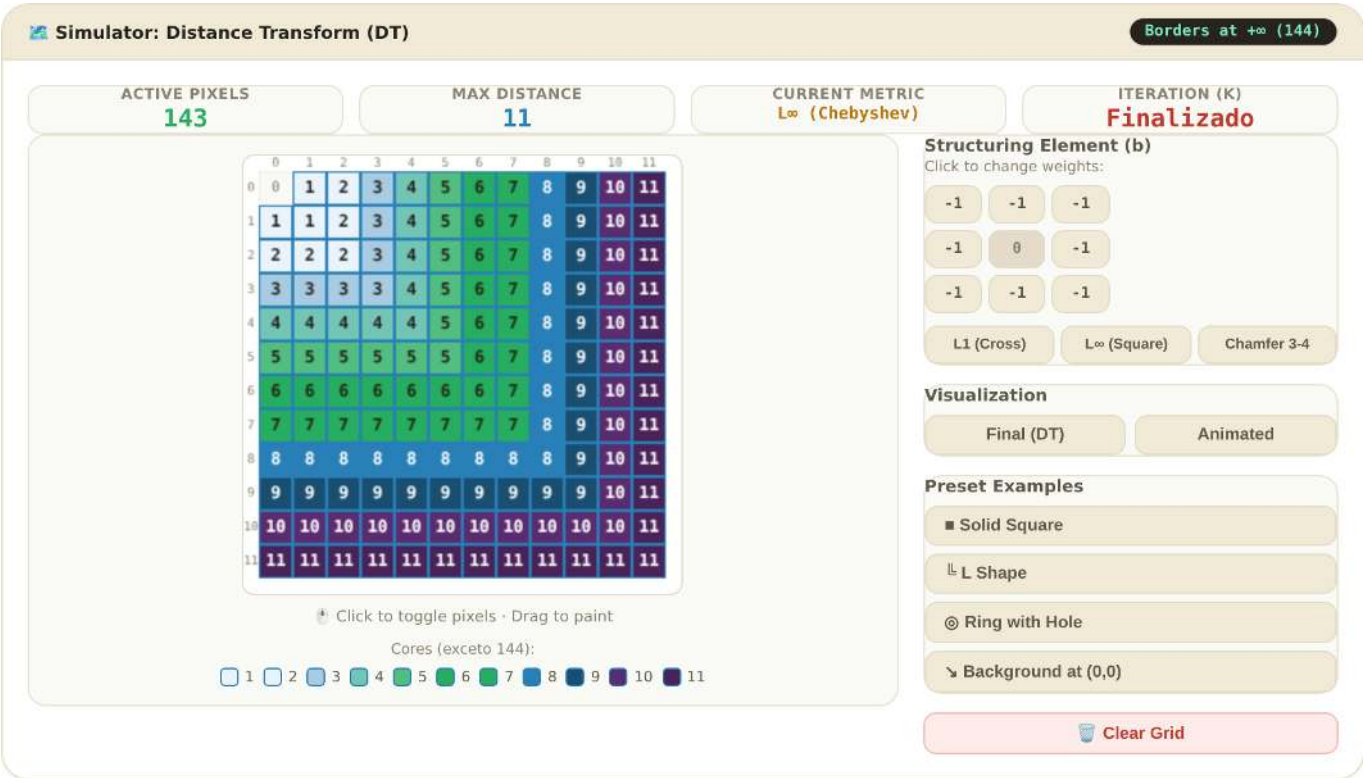
```
Max dist1 (erosions): 18 px
Max dist (L2)      : 12 px
```



Ponto Fixo Numérico: A distância emerge da propagação matemática do valor zero.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-alg-distancia2.html

Figure 4.18: Distance Transform Algorithm.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-04-distancia.html

Figure 4.19: Interactive simulator of the Distance Transform (DT) iterative via grayscale erosion. Pixels outside the image assume the maximum value (144), propagating costs from the internal background.

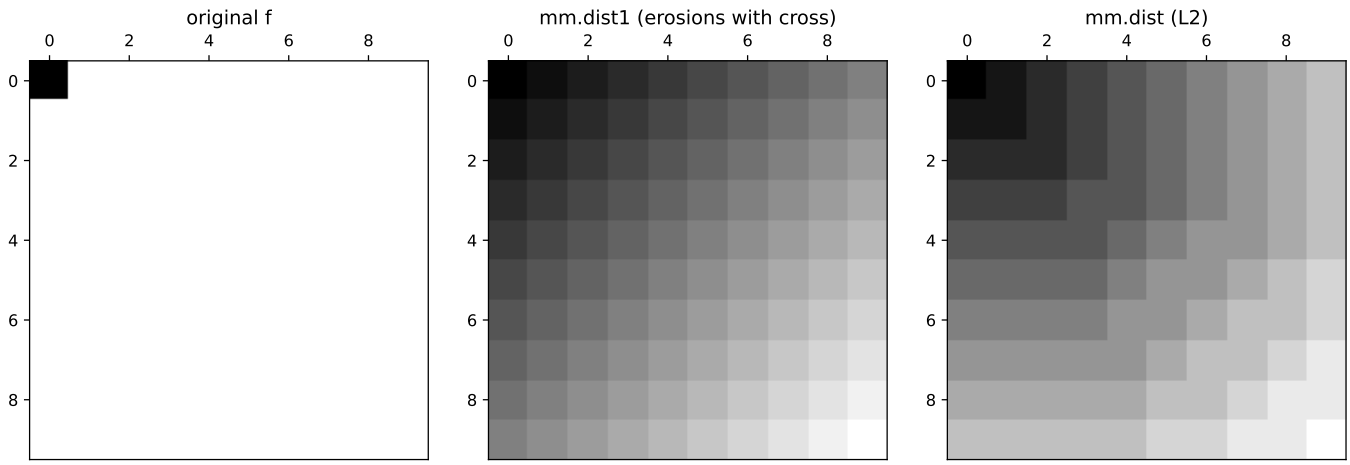
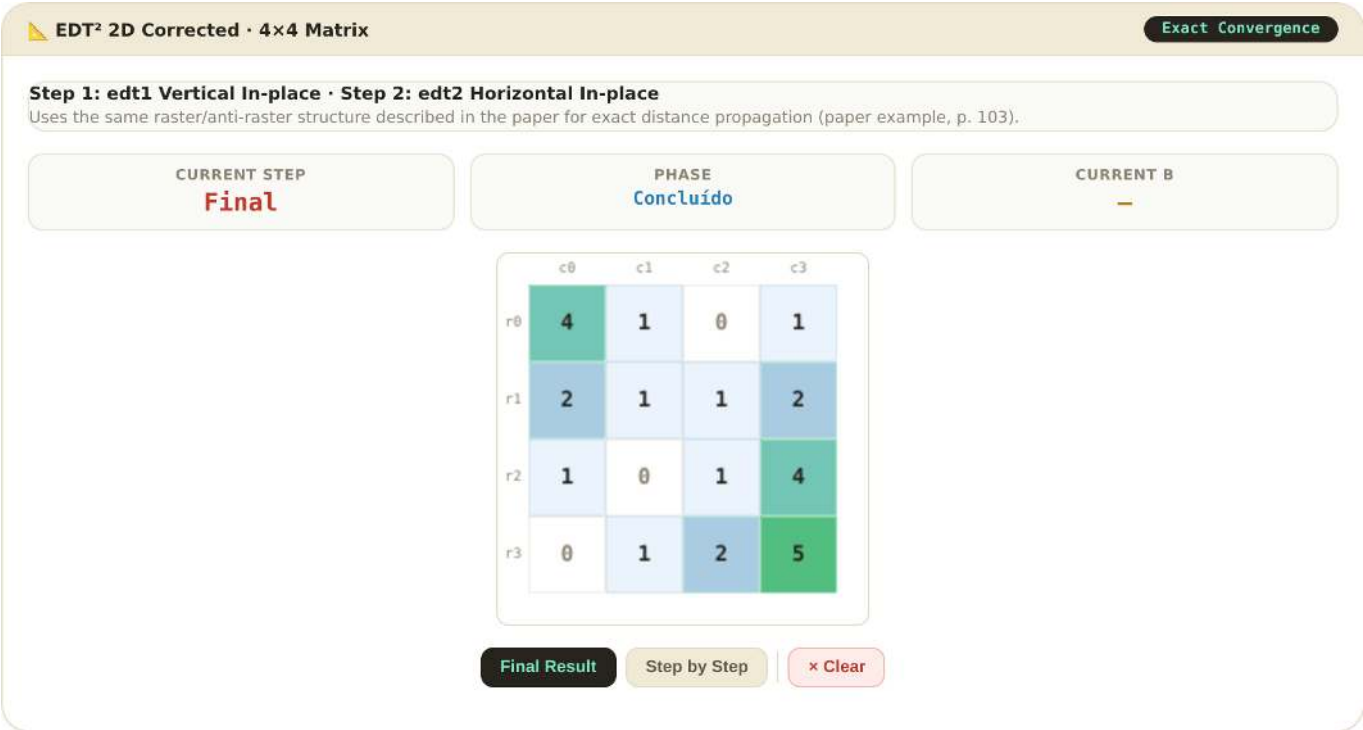


Figure 4.20: Distance Transform on a 10×10 binary image. Left: original (*foreground* = 255). Center: mm.dist1 iterative (erosions with cross). Right: mm.dist (L2).

Annotating numerical values directly over the pixels allows verifying how `dist1` propagates distances according to the metric induced by the structuring function used. In the case of the cross element with unit cost, the obtained values correspond to the *Manhattan* distance (L_1). Although `dist1` and `mm.dist` produce distinct numerical values because they adopt different metrics, both transforms preserve the topographical structure of the objects, causing their maxima to occur in similar central regions. This property justifies the use of `mm.dist` in practical applications, due to its high computational efficiency.

4.4.3 Euclidean Distance Transform in four steps

The simulator in Figure 4.21 implements the EDT algorithm from Lotufo (2001) in two stages. In the first stage, the function `edt1` performs a one-dimensional vertical transformation sequentially (*in-place*), traversing each column in *raster* (↓) and *anti-raster* (↑) order to compute distances in the vertical direction (two steps: South and North). In the second stage, the function `edt2` uses this result as input and performs horizontal propagation using queues: for each row of the matrix, two priority queues, `Eq` and `Wq`, are initialized by traversing the column indices in opposite directions (`Eq` from `W-1` to `1`, `Wq` from `2` to `W`), so that each updated pixel immediately enqueues its neighbors for reprocessing within the same round (two more steps: East and West). This queue mechanism allows successive erosions with increasing odd weights (`b = 1, 3, 5, ...`, incremented at each iteration of the outer loop) without propagation getting stuck on outdated values, since each round resolves the horizontal dependency chain completely before the next increment of `b`. The convergence of this propagation produces the Euclidean Distance Transform across the entire matrix, combining the vertical information obtained in `edt1` with the queue-based horizontal propagation performed in `edt2`.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-04-tde.html

Figure 4.21: Interactive 2D simulator (4x4) with strict synchronization of horizontal propagation queues (b) to obtain the exact convergence described in the paper.

4.4.3.1 Geodesic Distance Transform

The geodesic distance transform associates each pixel with the smallest distance to a marker, under the constraint imposed by a mask. Thus, propagation occurs exclusively through allowed pixels, preserving the connectivity of the domain.

Figure 4.22 illustrates this process in a maze: (a) the mask `g`; (b) the geodesic distance `D1` calculated from the entrance; (c) the distance `D2` calculated from the exit; and (d) the minimum path obtained from these two transforms.

The optimal path is determined by the sum of the distances (`D1 + D2`). The pixels belonging to the minimal trajectory are those for which this sum assumes its smallest value, defining a connection between entrance and exit with minimum geodesic length.

This principle allows solving mazes without the need to explicitly explore all possible routes. The solution

emerges directly from the propagation of distances in a restricted domain. This approach is particularly relevant in highly complex mazes, such as those constructed from quasicrystalline structures and Hamiltonian cycles described by Singh (2024). In Zampirolli (2025), this same formalism is employed to solve a complex maze; below, the method is illustrated in a simplified version of the problem.

```
import numpy as np

# 1 = corridor, 0 = wall
g = np.array([
 [0,1,0,0,0,0,0,0,0,0],
 [0,1,1,1,1,1,0,1,1,1],
 [0,0,0,0,0,1,0,1,0,1],
 [0,1,1,1,0,1,1,1,0,1],
 [0,1,0,1,0,0,0,0,0,1],
 [0,1,0,1,1,1,1,1,1,1],
 [0,1,0,0,0,0,0,0,1,0],
 [0,1,1,1,1,1,1,0,1,0],
 [0,0,0,0,0,0,1,1,1,0],
 [0,0,0,0,0,0,0,0,1,0]
], dtype=np.uint8)

# entrance marker
entrada = np.zeros_like(g, dtype=np.uint8)
entrada[0,1] = 1

# exit marker
saida = np.zeros_like(g, dtype=np.uint8)
saida[9,8] = 1

# geodesic distances
D1 = mm.gdist(g, entrada)
D2 = mm.gdist(g, saida)

# sum of distances
S = D1 + D2

# smallest valid value of the sum
dmin = np.min(S[S > 0])

# pixels belonging to an optimal path
caminho = (S == dmin)

print("Minimum geodesic distance:", dmin)

mm.show(
    [g, D1, D2, caminho],
    titles=[
        "Maze",
        "Distance from Entrance",
        "Distance from Exit",
        f"Shortest Path\n(d={dmin})"
    ],
    cols=4,
    figsize=(14,4),
    axis=True
)
```

```
Minimum geodesic distance: 15
```

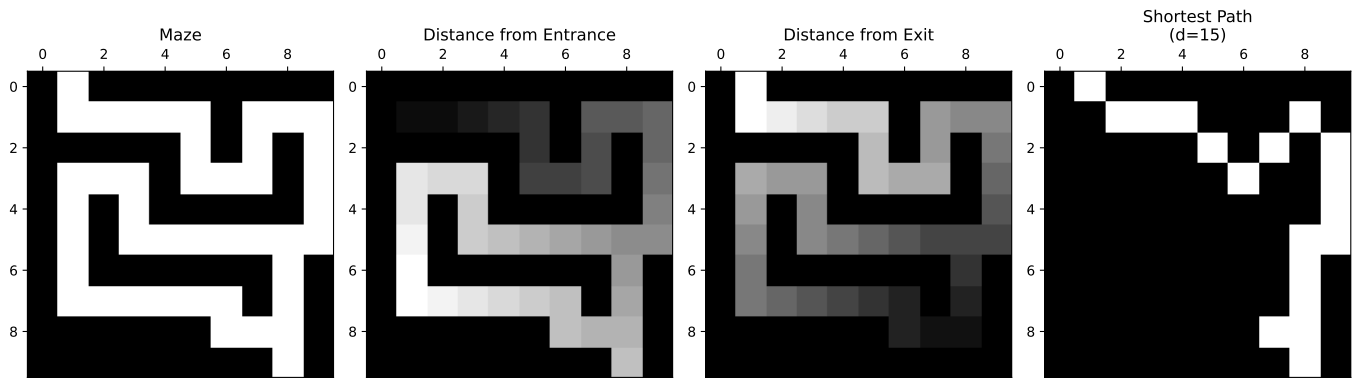


Figure 4.22: Shortest geodesic path in a maze. Geodesic distances are calculated from the entrance and exit using mm.gdist. Pixels whose sum of the two distances equals the minimum distance between markers belong to an optimal path.

4.4.4 Segmentation by *Watershed*

The ***Watershed*** algorithm interprets a grayscale image as a topographic surface, where high values correspond to mountains and low values correspond to valleys or catchment basins. In the context of marker-based segmentation, the maxima of the Distance Transform are often used to identify internal regions of objects, providing reliable seeds for the flooding process.

Segmentation is then performed through a conceptual simulation of progressive flooding from these markers. As the basins associated with different seeds expand, neighboring regions eventually come into contact. At that point, virtual barriers, known as *watershed lines*, are constructed, which then delimit the objects in the scene. This mechanism allows for the separation of adjacent or partially overlapping objects, even when they form a single connected component after thresholding.

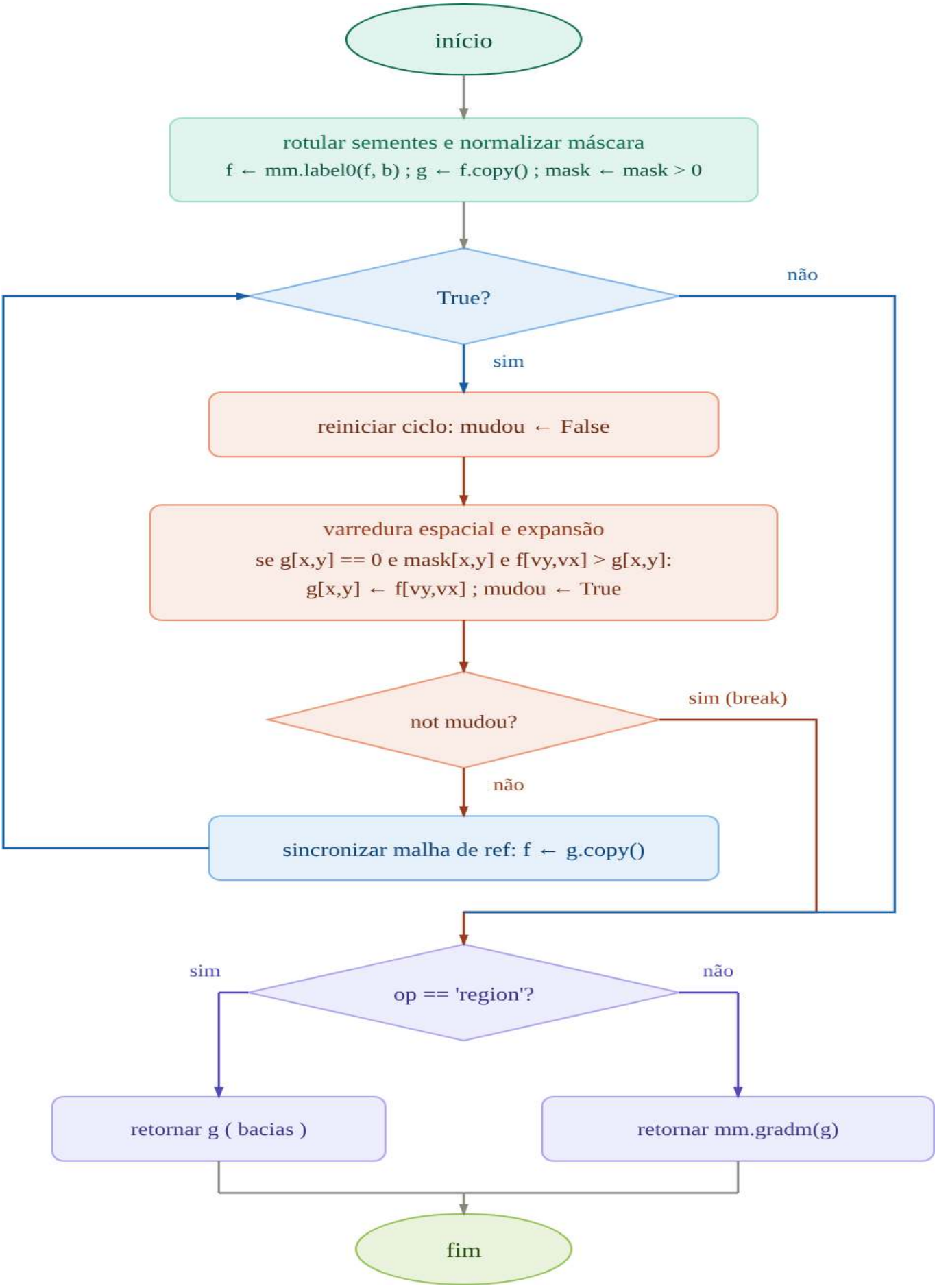
The didactic implementation presented in this chapter initially explores the concept of region growing confined by a binary mask, as detailed in the interactive algorithm of Figure 4.23.

> **ℹ Didactic version versus classical implementation**
>
> The `mm.watershed0` function does not implement the classical *watershed* algorithm. Its purpose is to illustrate, in a simplified manner, the propagation of markers through region growing, allowing one to visualize how different seeds compete for the occupation of available space. The growth is delimited by a binary support mask and monitored by a stagnation control, producing a result similar to a Voronoi partition restricted to the geometry of the input objects.
> The `mm.watershed` function, in turn, uses the optimized OpenCV implementation (`mm.watershed`), which performs flooding over a topographic surface defined by the input image. In this case, the propagation of markers is influenced by pixel values, causing separation lines to form naturally over the ridges of the relief.

Figure 4.24 presents an iterative simulator that illustrates the propagation of markers across the region of interest. Each marker acts as a flooding source that expands its area of influence until it encounters regions originating from other seeds. In the OpenCV algorithm, pixels belonging to the dividing lines are identified by the value `-1`, representing the boundaries between adjacent watersheds.

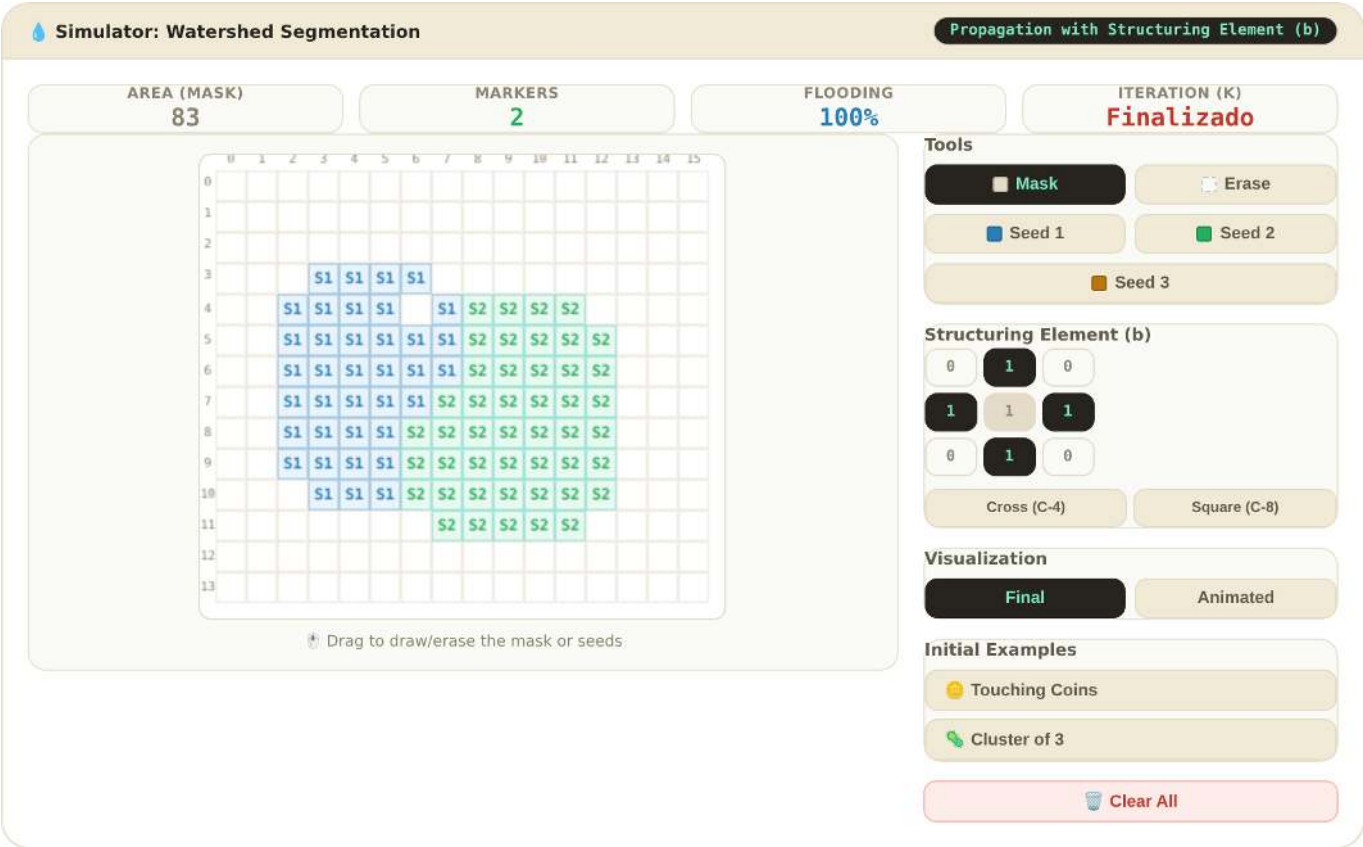
Morphological *Watershed* Pipeline

Marker-based *watershed* typically integrates into a broader segmentation workflow. In real images, preprocessing steps are often required to enhance contrast, reduce noise, and generate reliable markers. This complete workflow is summarized in Table 4.6.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-alg-watershed2.html

Figure 4.23: Didactic *Watershed* Algorithm by Region Growing Limited by Mask.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-04-watershed.html

Figure 4.24: Interactive simulator of the *Watershed* algorithm by morphological propagation. Draw the mask, position the markers and adjust the Structuring Element to observe the flooding. When basins meet simultaneously, the tie is resolved by randomly assuming one of the regions.

Table 4.6: Complete marker-based *watershed* pipeline for real images.

Step	Operation	Purpose
1	CLAHE + Smoothing	Contrast enhancement and noise reduction
2	Thresholding	Initial separation between object and background
3	Opening/Closing	Removal of noise and small imperfections
4	Mask Dilation	Identification of Sure Background
5	Distance Transform + Threshold	Identification of Sure Foreground
6	Uncertain Region	Difference between Sure Background and Sure Foreground
7	`mm.watershed`	Propagation of markers through the uncertain region

To focus exclusively on the concepts of Distance Transform, markers, and topographical flooding, the example in Figure 4.25 uses a synthetic binary image and adopts a simplified workflow, summarized in Table 4.7.

Table 4.7: Simplified pipeline used in the didactic example of Figure 4.25.

Step	Operation	Purpose
1	Distance Transform	Construction of the topographical surface
2	DT Threshold	Extraction of markers (Sure Foreground)
3	Dilation	Determination of Sure Background
4	Uncertain Region	Difference between background and markers
5	`mm.watershed`	Propagation of markers and generation of boundaries

```
# 1. Synthetic image and Distance Transform
f_sint = np.zeros((20, 20), dtype=np.uint8)
f_sint = mm.circle(f_sint, (6,  10), 5, 255, -1)
f_sint = mm.circle(f_sint, (14, 10), 5, 255, -1)
dist = mm.dist(f_sint)

# 2. Markers (distance peaks)
m = (dist > 0.8 * dist.max()).astype(np.uint8) * 255

# 3. Conditional execution of Watershed0 (m=markers first, mask=f_sint)
w_reg  = mm.watershed0(m, mask=f_sint, op='region')
w_line = mm.watershed0(m, mask=f_sint, op='line')

#w_reg  = mm.watershedB(m, mask=f_sint, op='region')
#w_line = mm.watershedB(m, mask=f_sint, op='line')

w_reg  = mm.watershed(m, mask=f_sint, op='region')
w_line = mm.watershed(m, mask=f_sint, op='line')

# 4. Displaying the results
mm.show([f_sint, dist, m, w_reg, w_line], cols=5, figsize=(16, 4),
```

```
        titles=["Original", "Distance", "Markers", "Regions", "Lines"])
```

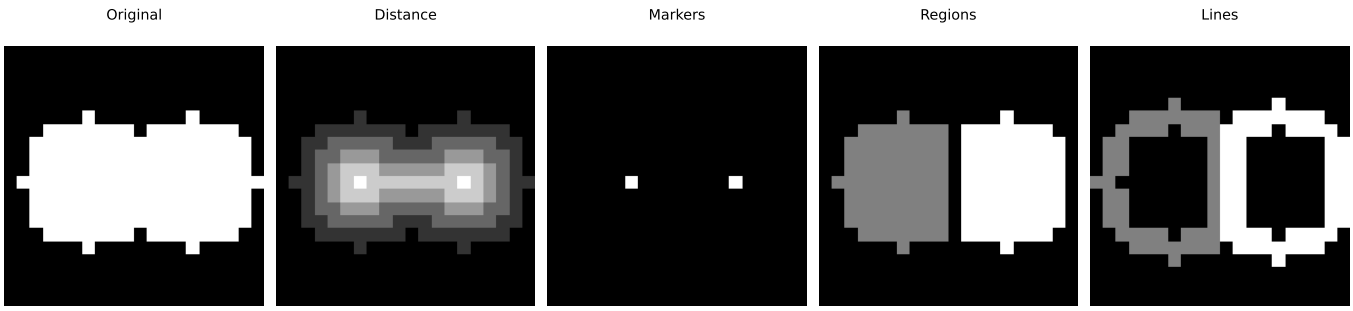


Figure 4.25: *Pipeline watershed* delimited by mask on a 20×20 binary image.

Application to overlapping coins:

```
img_base = img_coins_gray

# 1. Simulate overlapping/connected coins (using the base binary mask)
img_sobrepostas = mm.dil(img_final, mm.sebox(40))

# 2. Morphological opening to clean noise
opening = mm.open(img_sobrepostas, mm.sebox(2))

# 3. Distance Transform
dist = mm.dist(opening)
dist_vis=(255*(dist/dist.max())).astype(np.uint8) if dist.max() > 0 else dist.astype(np.uint8)

# 4. Safe peaks (Coin markers)
picos = (dist > 0.5 * dist.max()).astype(np.uint8) * 255

# 5. Watershed execution (Adjusted to use the new signature)
# We pass 'opening' directly in mask, as it delimits the expansion scope of the coins
ws_region = mm.watershed(picos, mask=opening, op='region')
ws_line   = mm.watershed(picos, mask=opening, op='line')

# 6. Object counting (Ignores background 0)
labels = np.unique(ws_region)
labels = labels[labels > 0]
print(f"Objects detected: {len(labels)}")

# 7. Final Annotation
img_annotated = cv2.cvtColor(img_base, cv2.COLOR_GRAY2BGR)

for idx, label_id in enumerate(labels):
    mask_reg = (ws_region == label_id).astype(np.uint8)
    if mask_reg.sum() < 500: continue

    cy, cx = np.mean(np.where(mask_reg), axis=1).astype(int)
    cv2.putText(img_annotated, str(idx + 1), (cx - 25, cy + 20),
                cv2.FONT_HERSHEY_SIMPLEX, 3.2, (0, 255, 0), 8, cv2.LINE_AA)

# Draws the separation lines in red
mask_ann = mm.dil(ws_line, np.ones((11, 11), np.uint8))
img_annotated[mask_ann > 0 ] = [255, 0, 0]

# Display
mm.show(
    [img_base, img_sobrepostas, dist_vis, picos, ws_region, img_annotated],
```

```
    titles=["Original", "Overlapping", "Distance", "Markers", "Watershed", "Annotated"],
    cols=3, rows=2, figsize=(12, 8)
)
```

```
Objects detected: 12
```

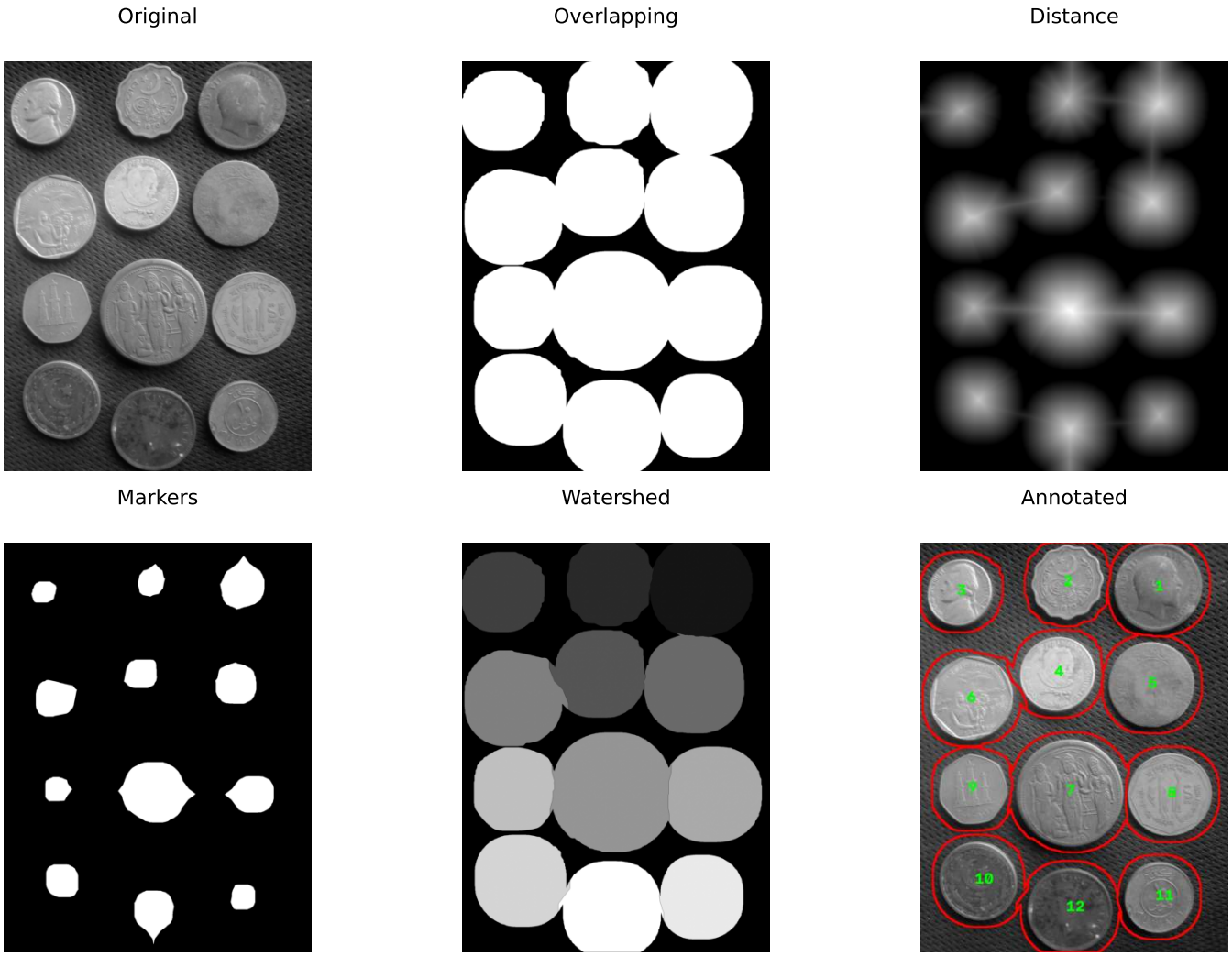


Figure 4.26: Watershed *Pipeline* for separating overlapping coins: from the dilated binary mask to the final annotated contours on the original image.

4.5 Shape Component Extraction and Descriptors

After segmentation and morphological refinement, the next step consists of identifying each object present in the image individually and extracting its geometric properties. This stage is fundamental for measurement, classification, and pattern recognition tasks.

A **connected component** is a maximal set of pixels belonging to the object that remain mutually connected according to a previously defined connectivity relation (4- or 8-connectivity). After labeling, each component receives a unique identifier, allowing its characteristics to be analyzed individually.

OpenCV offers two complementary approaches for this analysis, summarized in Table 4.8.

Table 4.8: Comparison between approaches based on connected components and contours.

	`connectedComponentsWithStats`	`findContours`
Returns	label per pixel and statistics per component	sequence of points describing the border
Direct descriptors	area, bounding box, and centroid	perimeter, shape, and hierarchy
Objects in contact	tends to merge connected regions	tends to produce a single external contour
Typical use	counting, filtering, and labeling	geometric analysis and shape descriptors

4.5.1 Labeling and Component Statistics

The function `mm.label0` simultaneously performs labeling of connected components and extraction of basic descriptors for each region. The operator returns:

- a label image (`labels`);
- geometric statistics (`stats`);
- centroid coordinates (`centroids`).

The statistics include area, width, height, and the position of the minimum bounding box aligned with the image axes.

Figure 4.27 illustrates the application of this operator after the coin segmentation pipeline. Each connected component receives a distinct color, and its area is annotated directly on the image.

```
# 1. Labeling and statistics
n, labels, stats, centroids = cv2.connectedComponentsWithStats(
    img_final, connectivity=8
)

# 2. Component coloring - FIXED palette (generated once with
# np.random.seed(4)) so the C++ track can reproduce it color by color without
# depending on the numpy generator. If n exceeds len(PALETTE), it cycles.
PALETA = np.array([
    [0, 0, 0], [224, 82, 193], [233, 155, 73], [190, 247, 244],
    [103, 94, 179], [51, 154, 59], [137, 96, 232], [250, 243, 205],
    [100, 141, 208], [228, 187, 163], [202, 108, 214], [131, 105, 104],
    [86, 153, 81]], dtype=np.uint8)
colors    = PALETA[np.arange(n) % len(PALETA)].copy()
colors[0] = [0, 0, 0]                          # black background
img_colored   = colors[labels]
img_annotated = img_colored.copy()

# 3. Descriptor table
print(f"Detected components (excluding background): {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", "Final Segmentation", "Connected Components", "Annotated Areas"],
    cols=4, figsize=(18, 6)
)
```

```
Detected components (excluding background): 12

  ID     Área     cx     cy      w      h
-----------------------------------------
   1   231969   1484    280    553    542
   2   158967    917    250    453    468
   3   139229    250    312    433    419
   4   175550    857    821    474    465
   5   222882   1442    889    539    531
   6   210043    316    977    527    516
   7   343376    934   1559    667    665
   8   213641   1555   1574    530    515
   9   147880    328   1543    432    438
  10   187280    363   2111    487    492
  11   150110   1487   2215    433    444
  12   215387    932   2292    528    522
```

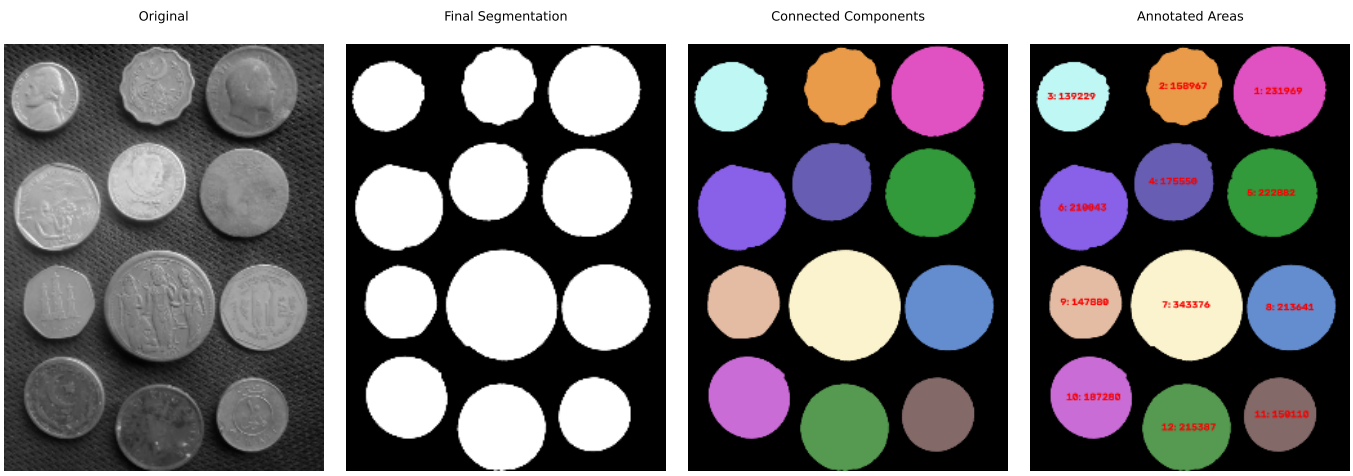


Figure 4.27: Connected components extracted after the CLAHE → Otsu → morphological cleanup *pipeline*. Each object is colored with a distinct color and annotated with its area in pixels.

4.5.2 Shape Descriptors

While `connectedComponentsWithStats` operates on regions, contour extraction acts directly on their boundaries. The operator returns, for each object, an ordered sequence of points that describes its contour.

From this representation, it is possible to compute geometric descriptors that are not directly provided by component labeling:

- **Area** (area of the contour)
- **Perimeter** (perimeter);
- **Circularity** ($C = \frac{4\pi A}{P^2}$, where A is the area and P is the perimeter);
- **Polygonal approximation** (polygonal approximation);
- **Convex hull** (*convex hull*);
- **Contour hierarchy**, allowing the representation of parent-child relationships between regions and their internal holes.

Circularity reaches a maximum value equal to 1 for a perfect circle and decreases as the object becomes more elongated or exhibits irregularities along its border.

Figure 4.28 presents the contours extracted from the segmented coins, along with the circularity values computed for each object.

```
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"Detected contours: {len(contornos)}")
print(f"\n{'ID':>4} {'Area':>8} {'Perimeter':>10} {'Circularity':>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=["Final Segmentation", "Contours", "Circularity"],
    cols=3, figsize=(18, 6)
)
```

```
Detected contours: 12

  ID     Area  Perimeter    Circularity
------------------------------------------
   1   214626     1769.6          0.861
   2   149480     1465.1          0.875
   3   186570     1654.9          0.856
   4   147252     1458.6          0.870
   5   212886     1756.3          0.867
   6   342424     2232.5          0.863
   7   209282     1767.7          0.842
   8   222108     1809.7          0.852
   9   174854     1616.4          0.841
  10   138602     1471.5          0.804
  11   158306     1538.7          0.840
  12   231181     1847.4          0.851
```

Figure 4.28: Contours extracted with *findContours*. Each coin is annotated with its circularity — values close to 1 confirm a circular shape.

4.5.3 Connection with Modern Object Detection

The descriptors extracted in the previous sections — especially *bounding boxes*, centroids, areas, and shape measures — establish a natural bridge between classical morphological segmentation and modern object detection systems. Although the techniques studied in this chapter use operations on pixels and segmented regions, many of the representations produced are directly compatible with the formats employed in contemporary computer vision models.

Deep learning-based detectors, such as the YOLO (*You Only Look Once*) family (REDMON, 2016), operate directly on color images and produce, for each detected object, a *bounding box* described by the center (cx, cy) and the dimensions (w, h), as well as a class and a confidence score. This representation shares the same basic geometric structure obtained by `connectedComponentsWithStats`, although it is produced by a learned model rather than by explicit segmentation.

Figure 4.29 illustrates how *bounding boxes* obtained by morphology can be exported in the YOLO format to compose datasets used in training or evaluating detectors.

```
# Recompute labels/statistics from the final segmentation
n, labels, stats, centroids = cv2.connectedComponentsWithStats(
    img_final, connectivity=8
)

H_img, W_img = img_coins_gray.shape
img_bbox = cv2.cvtColor(img_coins_gray, cv2.COLOR_GRAY2BGR)

CLASSE = 0   # 0 = coin (only category in this example)

print(f"{'cls':>4} {'cx_n':>8} {'cy_n':>8} {'w_n':>8} {'h_n':>8}  ← YOLO format")
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)

# Export annotation file in YOLO format
with open("moedas.txt", "w") as f:
    f.write("\n".join(yolo_linhas))
print("\nAnnotation saved to moedas.txt")

mm.show(
    [img_coins_gray, img_final, img_bbox],
    titles=["Original", "Final Segmentation", "Bounding Boxes (YOLO format)"],
    cols=3, figsize=(18, 6)
)
```

```
 cls     cx_n     cy_n      w_n      h_n  ← YOLO format
-------------------------------------------------------
   0   0.7753   0.1102   0.2880   0.2117
   0   0.4784   0.0984   0.2359   0.1828
   0   0.1331   0.1225   0.2255   0.1637
   0   0.4464   0.3217   0.2469   0.1816
   0   0.7523   0.3471   0.2807   0.2074
   0   0.1664   0.3812   0.2745   0.2016
   0   0.4862   0.6104   0.3474   0.2598
   0   0.8115   0.6154   0.2760   0.2012
   0   0.1724   0.6023   0.2250   0.1711
   0   0.1893   0.8258   0.2536   0.1922
   0   0.7763   0.8652   0.2255   0.1734
   0   0.4854   0.8953   0.2750   0.2039

Annotation saved to moedas.txt
```

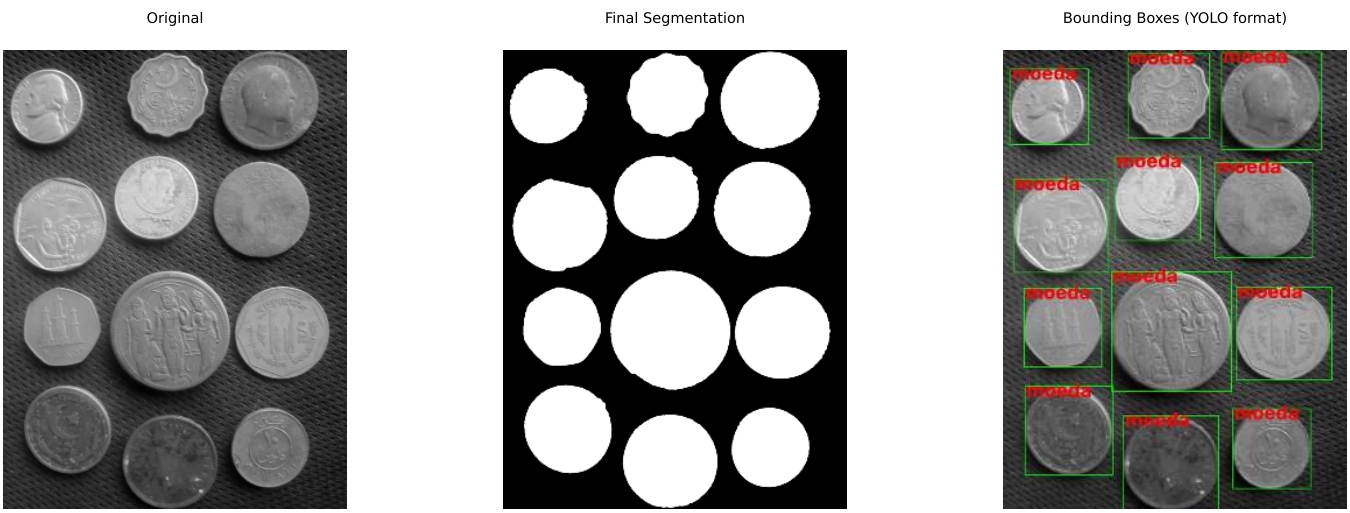


Figure 4.29: *Bounding boxes* derived from the connected components overlaid on the original image. The annotations are exported in YOLO format (*class cx cy w h*), with coordinates normalized to the range [0,1].

The YOLO format stores each object in a line containing five fields:

```
classe cx cy w h
```

where (cx, cy) represents the center of the *bounding box* and (w, h) its dimensions. All geometric values are normalized to the interval $[0, 1]$ with respect to the image's width and height. The **class** is an integer identifier associated with a category defined by the dataset (e.g., `0 → coin`). When there are multiple categories — such as gold coin (`0`), silver coin (`1`), and plastic disk (`2`) — one simply assigns the corresponding identifier to each object before export, maintaining exactly the same annotation format. In the previous example, this information was stored in the `moedas.txt` file.

The workflow presented in this chapter — segmentation → labeling → *bounding box* extraction — conceptually corresponds to the **annotation** (*labeling*) step employed in building training sets for modern detectors. Specialized tools, such as Label Studio and Roboflow, automate this process in complex images, but the fundamental logic remains the same: associating each object with a region of interest and a class. In controlled scenarios, with a uniform background and well-separated objects, morphological techniques can even generate annotations automatically or serve as a starting point for manual labeling, significantly reducing the effort of dataset construction. In more complex real-world applications, however, human validation remains necessary to ensure the quality of annotations.

Evaluation: IoU (*Intersection over Union*)

A simple way to evaluate the quality of a segmentation is to compare it against a reference mask (*ground truth*). The most widely used metric for this purpose is **IoU** (*Intersection over Union*):

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} \tag{4.16}$$

where A represents the segmentation produced by the algorithm and B the reference segmentation. The IoU value ranges between 0 and 1. The higher the value, the greater the overlap between the masks. An IoU of 1 indicates perfect correspondence between the obtained segmentation and the reference. The same metric is also extensively used in object detection, applied to predicted and annotated *bounding boxes*. In this field, IoU values greater than 0.5 are often adopted as the minimum criterion to consider a detection correct.

Beyond Morphology

The techniques studied in this chapter segment objects by exploiting spatial connectivity, morphological operations, and topographic relief. However, there are alternative approaches based on feature clustering, such as the *k-means* algorithm, Gaussian mixture models (GMM), and more recent methods based on deep learning. These techniques will be revisited in Part II of the book, dedicated to Computer Vision.

4.6 Summary

This chapter presented the main segmentation and mathematical morphology techniques, concluding the study of DIP in the spatial domain.

- **Preprocessing and thresholding:** The combination of adaptive CLAHE equalization and Otsu's method proved effective in inducing bimodal separation in the histogram and simplifying the binarization of images with non-uniform illumination.
- **Erosion and dilation:** Fundamental morphological operators based on the search for local minima and maxima in a neighborhood defined by the structuring element B. These are dual operators by complement and were implemented using the functions `mm.ero` and `mm.dil`.
- **Opening and closing:** Compositions of erosion and dilation that allow removing noise, smoothing contours, and filling small gaps while preserving the overall structure of objects.

- **Morphological reconstruction:** An iterative geodesic process that propagates a marker within the limits imposed by a mask, forming the basis of operators such as `mm.clohole` and `mm.edgeoff`.
- **Binary cleaning pipeline:** A consolidated workflow composed of CLAHE → Otsu → opening → `mm.clohole` → restricted opening → `mm.edgeoff`, producing suitable masks for quantitative analysis.
- **Grayscale morphology:** An algebraic extension based on weighted minima and maxima, enabling operators such as morphological gradient and *top-hat* filters for enhancing local structures.
- **Distance Transform and Watershed:** The Distance Transform allowed generating automatic markers for the *watershed* algorithm, enabling the separation of adjacent or partially overlapping objects.
- **Connected components and descriptors:** Region labeling (`mm.label0`) and contour extraction (contour extraction) allowed computing geometric descriptors such as area, centroid, perimeter, circularity, and bounding boxes.
- **Connection with Modern Computer Vision:** The bounding boxes extracted by morphology were exported in YOLO format, highlighting the link between classical segmentation techniques and modern object detection systems.

Chapter 5 will introduce processing techniques in the **frequency domain**, addressing the **Fourier Transform**, spectral filtering, and the fundamentals of **image compression**, including DCT, JPEG, and wavelets.

4.7 🤖 Using Gemini Notebook as a Complementary Tutor

In this edition, the use of **Gemini Notebook** is encouraged as a complementary learning tool. Based on artificial intelligence, the system uses exclusively the documents provided by the author as its source of knowledge, producing responses aligned with the content and approach adopted throughout this chapter.

🎓 Study with the Intelligent Tutor

🚀 ACCESS GEMINI NOTEBOOK: CHAPTER 04

🌐 **Language and Programming Language**

The project for this chapter in Gemini Notebook was built using only the text in **Portuguese** and the code examples in **Python**. If you are studying from the English or French edition, or following the C++ track, the tutor's responses may not correspond exactly to the version you are reading.

⚠️ **Notice Regarding AI-Generated Content**

Although it is a valuable study support tool, Gemini Notebook may occasionally produce incomplete, inaccurate, or incorrect responses. It is recommended to validate the information by consulting the chapter material, books, scientific articles, and other reliable academic sources. Whenever possible, run and experiment with the practical examples presented throughout the text to consolidate your understanding of the concepts.

4.8 Exercise List

1. **(10%)** Manually implement Otsu's criterion without using `mm.threshold`. Compute the interclass variance $\sigma_B^2(T)$ for all thresholds $T \in [0, 255]$ using `mm.hist`, identify the optimal threshold T^* and compare the result with the value obtained by OpenCV. Plot σ_B^2 as a function of T and highlight the maximum point.

2. **(15%)** Apply adaptive thresholding with block sizes of 11, 31, and 51 to an image containing non-uniform illumination. Compare the results with global Otsu thresholding and discuss the advantages and limitations of each approach.

3. **(15%)** Run the watershed *pipeline* on the coins image, varying the threshold applied to the Distance Transform (0.3, 0.5, and 0.7 times the maximum value). Explain how this parameter influences marker generation, the separation of adjacent objects, and the occurrence of over-segmentation.
4. **(15%)** Using `mm.drawImg`, construct a step-by-step visual demonstration of the erosion of a 7×7 binary image with a 3×3 square structuring element. For each analyzed position, indicate whether the structuring element is fully contained within the object and justify the value assigned to the output pixel.
5. **(15%)** Experimentally demonstrate the duality between erosion and dilation by verifying the identity $(A \ominus B)^c = A^c \oplus \hat{B}$ using `mm.ero`, `mm.dil`, and `mm.bnot`. Compute the pixel-by-pixel difference between the two sides of the equation and present the result using `mm.histImg` or equivalent visualization.
6. **(15%)** Manually implement the morphological gradient using only `mm.ero` and `mm.dil`, comparing the result with `mm.gradm(img, B)`. Evaluate the effect of different structuring elements (3×3 square, 5×5 disk, and 1×9 line) on edge detection.
7. **(15%)** Construct a complete *pipeline* for counting and classifying coins by size (small, medium, and large) using area and circularity as descriptors. Manually generate a reference mask (*ground truth*) and compute the IoU metric (*Intersection over Union*) to assess segmentation quality. Present the results in a table and through visualizations produced with `mm.show`.

Chapter References

The theoretical foundation of this chapter is based on the following works:

- Gonzalez (2018) for the concepts of segmentation, Otsu thresholding, Distance Transform, *watershed*, mathematical morphology, and shape descriptors.
- Matheron (1975) and Serra (1982) for the original theoretical foundation, algebraic formulation, and development of Mathematical Morphology.
- Szeliski (2022) for region-based segmentation, connected component labeling, marker-controlled *watershed*, and segmentation evaluation using the IoU metric.
- Bradski (2008) for the practical use of the OpenCV library, including functions such as `mm.dist`, `mm.watershed`, `mm.label0`, and contour extraction.
- Redmon (2016) for an introduction to modern detectors of the YOLO family and their relationship with geometric descriptors such as *bounding boxes* extracted by segmentation.
- Singh (2024) for the construction of complex mazes based on Hamiltonian cycles over quasicrystalline mosaics, used as an application example of the Geodesic Distance Transform and pathfinding algorithms.
- Zampirolli (2025) for the implementation of morphological operators, geodesic transforms, and maze solving by distance propagation in restricted domains.

---

cap04/04/images/gitlabbadge.png

4.9 💻 Practical Part with Programming Exercises

This list transforms the concepts from Chapter 4 into a practical track on segmentation and mathematical morphology. The programming exercises begin with thresholding and advance to labeling and component descriptors, always using small matrices so that each pixel can be verified by hand.

! Common rule for morphological programming exercises

In neighborhood operations, **do not apply padding**. For each pixel, evaluate only the positions of the structuring element that fall within the image domain. This is the same idea behind the didactic implementations in `morph.py`, such as `mm.dil0`, `mm.ero0`, `mm.dil1`, and `mm.label0`: the neighborhood

is clipped by the valid domain of the image.

🎯 Objective of this Notebook

The notebook allows you to develop, validate, organize, and test solutions for **Programming Exercises (EPs)** in interactive environments, such as Colab, using the same test cases as Moodle, and copying them there only when recording the official grade.

Download

Download `morph.py` and `testsuite.py` by running the cell below:

```
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup(testsuite=True)
from morph import mm
from testsuite import TestSuite
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Executing Tests

To evaluate the tests, run `TestSuite("EP04_01.extension").run()` in a new cell, replacing the extension with that of the language used (`.py`, `.java`, `.c`, `.cpp`, `.js`, or `.r`). The system downloads the test cases from GitHub, runs the program, and computes the grade automatically.

To test Python code directly, without saving a file, use `run_code(code)` by passing the code as a *string* in a variable `code`:

```
code = """
from morph import mm
# ... your code here ...
"""
TestSuite("EP04_01").run_code(code)
```

4.9.1 EP04_01 Global Thresholding with a Fixed Threshold

In **document scanners** and **barcode reading systems**, the first processing step is always to separate what is "object" (ink, text, bars) from what is "background" (paper, packaging). **Global thresholding** does exactly this: it compares each pixel to a single threshold T and decides, in real time, whether it belongs to the light class or the dark class. It is the simplest segmentation operator—and yet, it underlies a large portion of industrial visual inspection *pipelines*. See Figure 4.30 for a simulation of this EP.

4.9.1.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Threshold:** Read the integer T (decision threshold).
3. **Data:** Read the integer values of the original matrix row by row.
4. **Mapping:** For each pixel p, compute the new value using the equation:

$$p' = \begin{cases} 255, & \text{if } p > T \\ 0, & \text{if } p \leq T \end{cases}$$

5. **Output:** Display the binarized matrix with dimensions $L \times C$.

4.9.1.2 📌 Computational Constraints

- **Binarization:** The output contains **only** the values 0 or 255.
- **Strict comparison:** The criterion uses $> T$ (pixels equal to T become background).
- **Type:** The final result must be an integer.
- **Note:** This EP follows the OpenCV convention (`cv2.THRESL_BINARY`): only pixels with value **greater than** T become white (`255`); pixels with value **equal to** T remain black (`0`).

4.9.1.3 🧠 Theoretical Foundation

Parameter	Type	Visual Impact
T **small**	Integer	Most pixels become white
T **large**	Integer	Most pixels become black
T **well-chosen**	Integer	Clearly separates object and background

4.9.1.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer T.
- Following lines: Integer elements of the original matrix.

Output:

- Binarized matrix with L rows and C columns, values 0 or 255 separated by spaces.

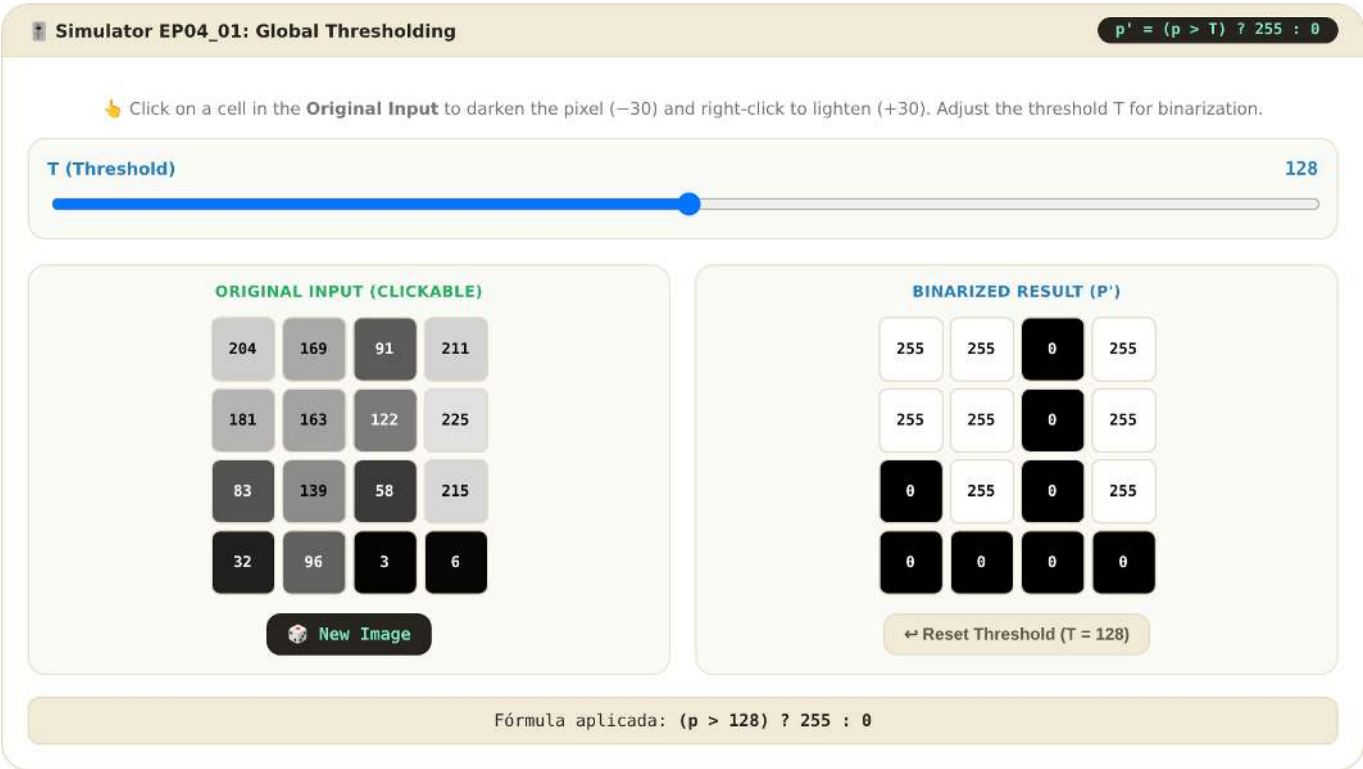
4.9.1.5 📌 Examples

Input	Output	Observation
2 4 100 0 99 100 180 255 30 120 80	0 0 0 255 255 0 255 0	$T = 100$: only pixels with value greater than 100 become white; therefore, 99 and 100 become black.
1 3 0 0 50 255	0 255 255	$T = 0$: only pixels with value strictly greater than 0 become white.

```
%%writefile EP04_01.py
# Python code
```

```
Overwriting EP04_01.py
```

```
TestSuite("EP04_01.py").run()
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-ep0401-limiar.html

Figure 4.30: EP04_01 Simulator: Global Thresholding by Fixed Threshold (p' = (p > T) ? 255 : 0)

```
<IPython.core.display.HTML object>
```

4.9.2 EP04_02 📊 Otsu's Automatic Thresholding

Manually choosing the threshold T works when illumination is stable, but in **digital microscopy** and **blood smear inspection**, each sample has different contrast — a fixed threshold would fail from image to image. **Otsu's method** solves this by autonomously finding the threshold that **maximizes the statistical separation** between the two pixel classes, making segmentation automatic and adaptive. See Figure 4.31 for a simulation of this EP.

4.9.2.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Data:** Read the integer values of the original matrix row by row.
3. **Histogram:** Build the histogram $h[i]$, $i = 0, \ldots, 255$, counting how many pixels have value i.
4. **Threshold search:** For each candidate T from 1 to 255, compute the **between-class variance**:

$$\sigma_B^2(T) = \frac{n_0 \cdot n_1}{N^2}(m_0 - m_1)^2$$

   where n_0, n_1 are the numbers of pixels with values $< T$ and $\geq T$, m_0, m_1 are their means, and $N = L \times C$.

5. **Selection:** The optimal threshold T^* is the one that maximizes $\sigma_B^2(T)$ (in case of a tie, keep the **first** one found).
6. Application: Binarize the image using T*, applying:

$$p' = \begin{cases} 255, & \text{if } p > T^* \\ 0, & \text{if } p \leq T^* \end{cases}$$

4.9.2.2 📌 Computational Constraints

- **Valid candidates:** Ignore T that leaves $n_0 = 0$ or $n_1 = 0$ (empty class).
- **Tie-breaking:** Always keep the **first** T that reached the maximum value of σ_B^2.
- **Type:** T^* and the output matrix must be integers.
- **OpenCV convention:** The binarization follows `cv2.THRESL_BINARY`; pixels with value exactly equal to T^* become black.

4.9.2.3 🧠 Theoretical Background

Concept	Meaning	Impact
High $\sigma_B^2(T)$	Classes well separated at T	T is a good threshold candidate
Bimodal histogram	Two distinct "peaks"	Otsu finds the valley between them
Unimodal histogram	A single "peak"	Otsu still chooses *some* T, but the segmentation is unreliable

4.9.2.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Following lines: Integer elements of the original matrix.

Output:

- Binarized matrix with L rows and C columns, values 0 or 255.

4.9.2.5 📌 Examples

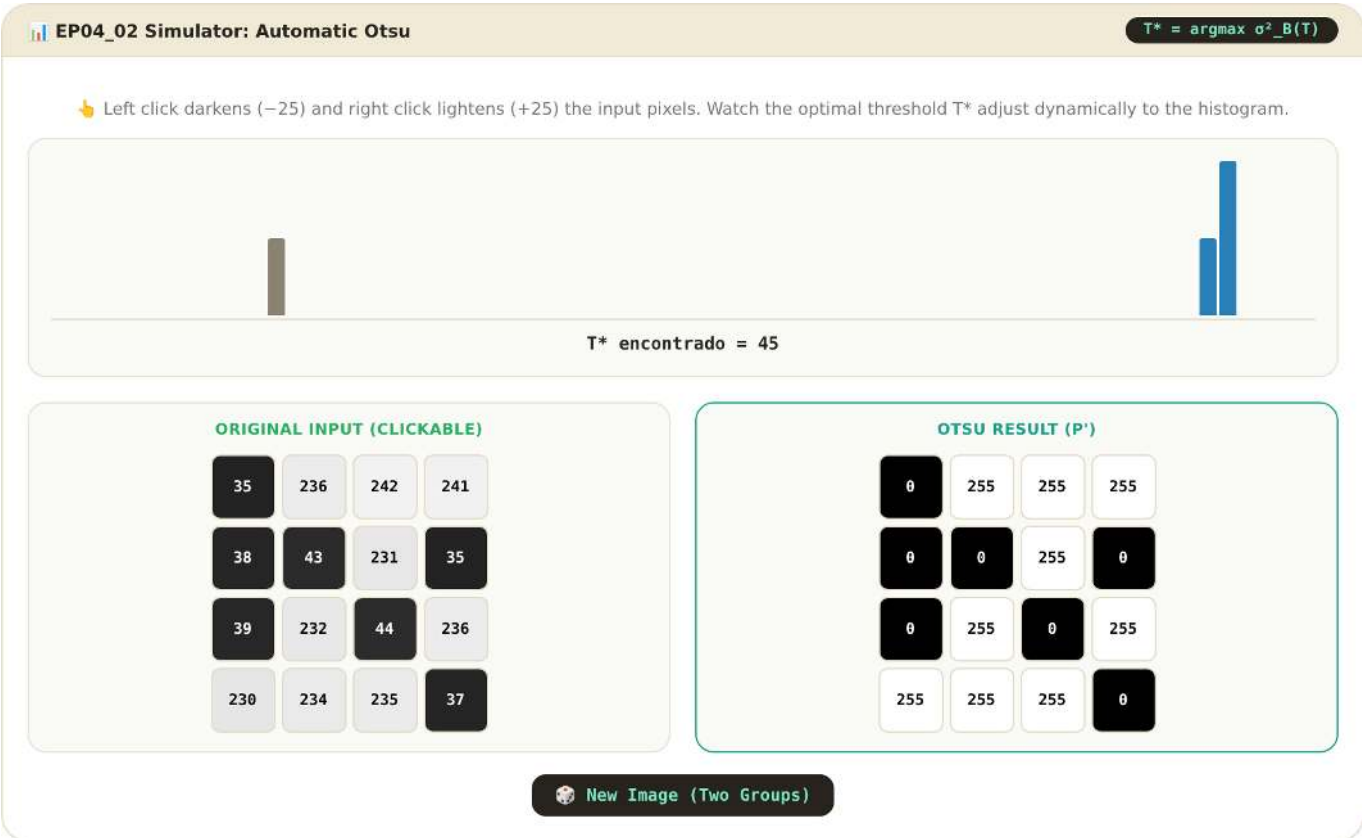
Input	Output	Observation
4 4 12 12 12 200 12 12 200 200 12 200 200 200 200 200 200 200	0 0 0 255 0 0 255 255 0 255 255 255 255 255 255 255	Clear bimodal histogram: 12 and 200
1 2 10 250	0 250	Only two values: T^* falls on the largest one

```
%%writefile EP04_02.py
# Python code
```

```
Overwriting EP04_02.py
```

```
TestSuite("EP04_02.py").run()
```

```
<IPython.core.display.HTML object>
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-ep0402-otsu.html

Figure 4.31: EP04_02 Simulator: Automatic Otsu Thresholding (T* = argmax ²_B(T))

4.9.3 EP04_03 🌱 Flat Binary Dilation (mm.dil0)

In **particle microscopy** and in **OCR of worn license plates**, thin or discontinuous traces need to be "thickened" for recognition to work. **Morphological dilation** does exactly that: it expands bright regions using a structuring element B — the same operation implemented in `morph.py` as `mm.dil0(f, B)`, used when B is **flat** (without weights, only 0/1). See Figure 4.32 for a simulation of this EP.

4.9.3.1 📋 Implementation Guidelines

1. **Image dimensions:** Read the integers L (rows) and C (columns) from f.
2. **Dimensions of B:** Read the integers L_B (rows) and C_B (columns) of the structuring element.
3. **Structuring element:** Read the matrix B with values 0 or 1, row by row.
4. **Data:** Read the matrix f (the original image), row by row.
5. **Reflection:** Construct B_{ref}, the version of B reflected by 180° (rows and columns reversed) — exactly as `mm.dil0` does internally.
6. **Neighborhood without padding:** For each pixel (y, x), traverse the positions (by, bx) of B_{ref} centered at (y, x), using the offset

$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\tfrac{L_B}{2} + 0{,}5, \quad o_x = -\tfrac{C_B}{2} + 0{,}5$$

   Discard every (v_y, v_x) outside $[0, L) \times [0, C)$ — **do not pad with zeros**.
7. **Mapping:** Compute each output pixel as the **maximum** between $f(y, x)$ and all valid $f(v_y, v_x)$ whose corresponding position in B_{ref} is 1:

$$g(y, x) = \max\Big(f(y, x), \max_{\substack{(v_y, v_x) \text{ valid} \\ B_{ref}(by, bx)=1}} f(v_y, v_x)\Big)$$

8. **Output:** Display the matrix g with dimensions $L \times C$.

4.9.3.2 📌 Computational Constraints

- **No padding:** Never invent neighbors outside the image; use only those that actually exist.
- **Mandatory reflection:** B must be reflected before being applied (this is what distinguishes `mm.dilO` from a simple maximum search).
- **Edge robustness:** If no valid position of $B_{ref} = 1$ falls within the domain for a given pixel, it **maintains its original value**.

4.9.3.3 🧠 Theoretical Foundation

Concept	Meaning	Visual Impact
Dilation	$g \geq f$ always (extensive)	Bright regions grow, dark holes shrink
Larger B	Wider neighborhood	More aggressive growth
Reflection of B	$B_{ref}(y, x) = B(-y, -x)$	Ensures the formal Minkowski definition of dilation

4.9.3.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer L_B.
- Line 4: Integer C_B.
- Next L_B lines: integer elements (0 or 1) of the matrix B.
- Next L lines: integer elements of the matrix f.

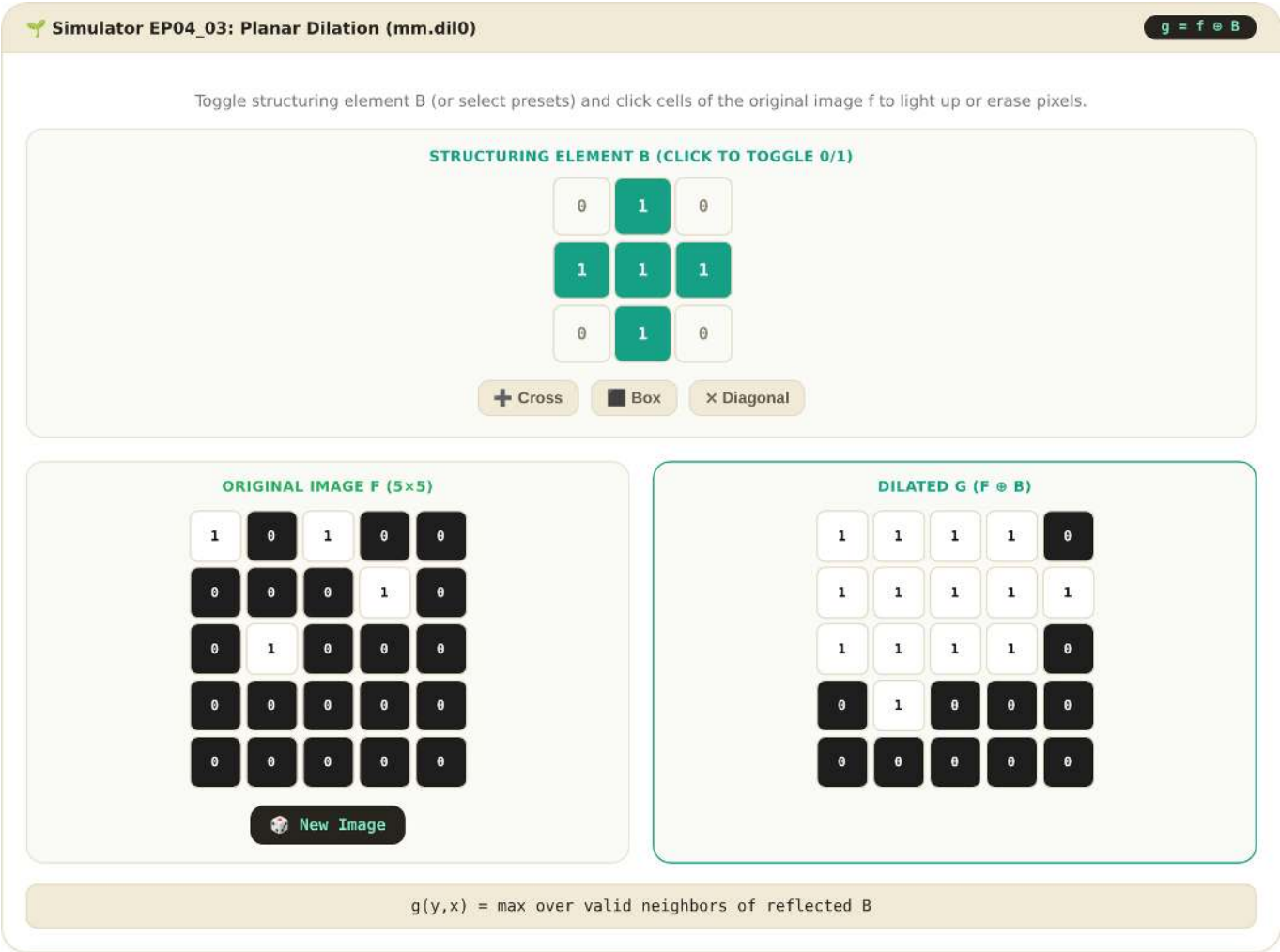
Output:

- Matrix g in L rows and C columns, integer values separated by spaces.

4.9.3.5 📌 Examples

Input	Output	Remark
3 3 3 3 0 1 0 1 1 1 0 1 0 0 0 0 0 9 0 0 0 0	0 9 0 9 9 9 0 9 0	Symmetric cross B: isolated point expands into a cross
1 4 1 3 1 1 1 10 200 5 80	200 200 200 80	Horizontal B: each pixel "pulls" the maximum of row neighbors

```
%%writefile EP04_03.py
# Python code
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-ep0403-dilatacao.html

Figure 4.32: Simulator EP04_03: Binary Dilation (g = f B)

```
Overwriting EP04_03.py
```

```
TestSuite("EP04_03.py").run()
```

```
<IPython.core.display.HTML object>
```

4.9.4 EP04_04 🪨 Binary Erosion by Flat Structuring Element (mm.ero0)

If dilation thickens, **erosion** thins. In **cell counting systems**, it is used to **separate touching cells**: by "eating away" the borders of each region, thin connections between objects disappear even before any counting is performed. In `morph.py`, this is the operation `mm.ero0(f, B)` — the exact **dual** of dilation, and the only one of the two that does **not** reflect the structuring element. See Figure 4.33 for a simulation of this exercise.

4.9.4.1 📋 Implementation Guidelines

1. **Image dimensions:** Read the integers L (rows) and C (columns) from f.
2. **Dimensions of B:** Read the integers L_B (rows) and C_B (columns) of the structuring element.
3. **Structuring element:** Read the matrix B with values 0 or 1, row by row.
4. **Data:** Read the matrix f (the original image), row by row.
5. **Neighborhood without padding (no reflection!):** For each pixel (y, x), traverse the positions (by, bx) of B **in the original order** (without reflecting), using the same offset as in EP04_03:

$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\tfrac{L_B}{2} + 0{,}5, \quad o_x = -\tfrac{C_B}{2} + 0{,}5$$

Discard every (v_y, v_x) outside $[0, L) \times [0, C)$. 6. **Mapping:** Compute each output pixel as the **minimum** between $f(y, x)$ and all valid $f(v_y, v_x)$ whose corresponding position in B equals 1:

$$g(y, x) = \min\Big(f(y, x), \min_{\substack{(v_y, v_x) \text{ valid} \\ B(by, bx)=1}} f(v_y, v_x)\Big)$$

7. **Output:** Display the matrix g with dimensions $L \times C$.

4.9.4.2 📌 Computational Constraints

- **No reflection:** Unlike dilation, B is used **exactly as read** — reflecting it here would be a serious conceptual error.
- **No padding:** Neighbors outside the image are simply ignored, never treated as 0.
- **Edge robustness:** If no valid position of $B = 1$ falls within the domain, the pixel retains its original value.

4.9.4.3 🧠 Theoretical Background

Concept	Meaning	Visual Impact
Erosion	$g \leq f$ always (anti-extensive)	Bright regions shrink, point noise disappears
Duality	$\text{ero}(f, B) = -\text{dil}(-f, B_{ref})$	Erosion and dilation are mathematical "mirrors"
Larger B	More aggressive erosion	Thin objects disappear completely

4.9.4.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer L_B.
- Line 4: Integer C_B.
- Next L_B lines: integer elements (0 or 1) of matrix B.
- Next L lines: integer elements of matrix f.

Output:

- Matrix g in L rows and C columns, integer values separated by spaces.

4.9.4.5 📌 Examples

Input	Output	Observation
3 3 3 3 0 1 0 1 1 1 0 1 0 9 9 9 9 0 9 9 9 9	9 0 9 0 0 0 9 0 9	The central "hole" (0) propagates in a cross pattern
1 4 1 3 1 1 1 10 200 5 80	10 5 5 80	Horizontal B: each pixel "pulls" the minimum of its row neighbors

```
%%writefile EP04_04.py
# Python code
```

```
Overwriting EP04_04.py
```

```
TestSuite("EP04_04.py").run()
```

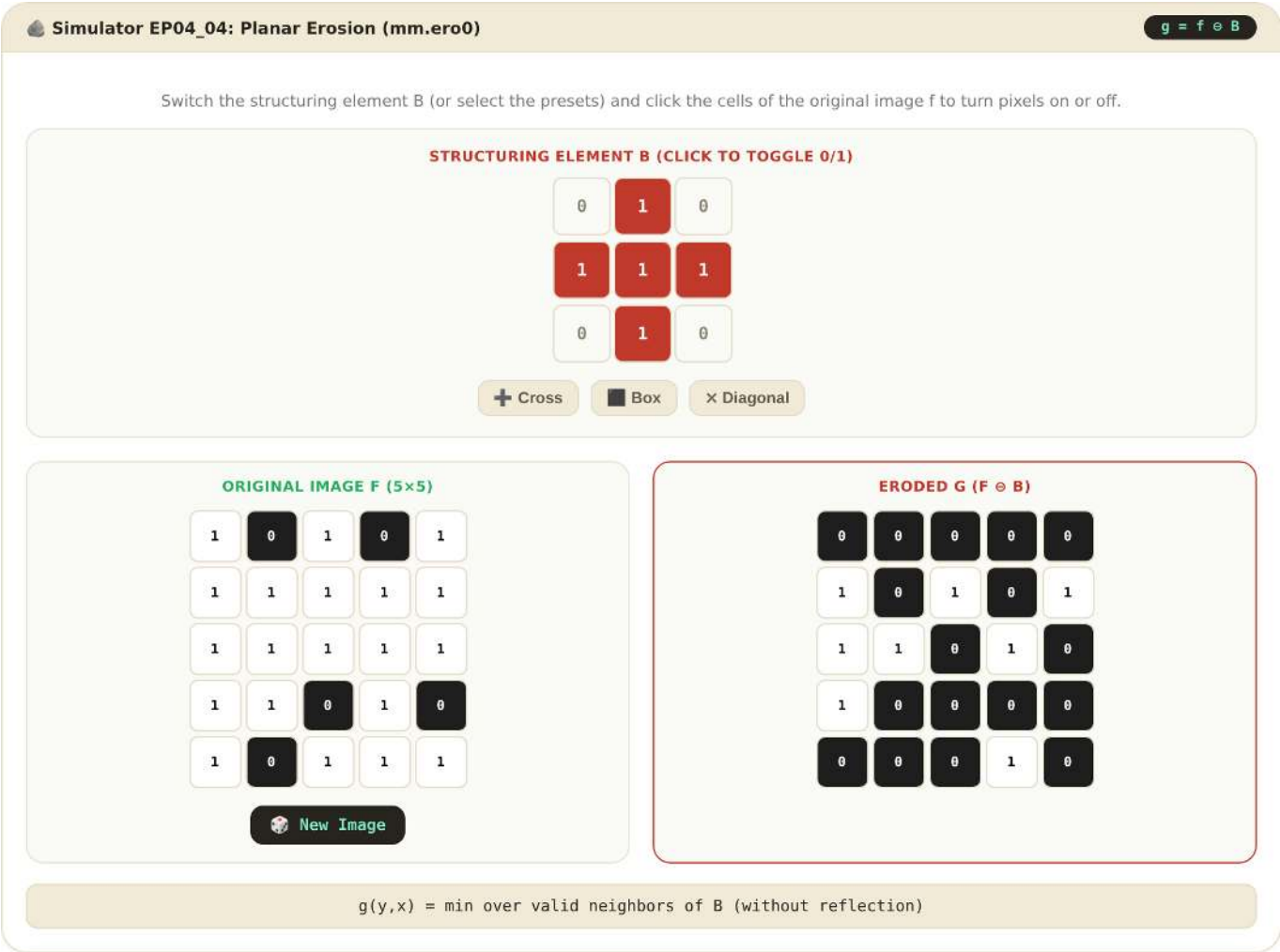
```
<IPython.core.display.HTML object>
```

4.9.5 EP04_05 🧹 Morphological Opening (Noise Removal)

Images captured by **low-cost sensors**, such as those on agricultural drones, often come dotted with small noise points—isolated pixels that represent nothing real. Applying erosion followed by dilation with the **same** structuring element produces the **opening**: it "cleans" points and thin protrusions, but returns the main object to nearly its original size. This is the classic combination used in **satellite image preprocessing** before any planted-area counting. See Figure 4.34 for a simulation of this EP.

4.9.5.1 📋 Implementation Guidelines

1. **Image dimensions:** Read integers L (rows) and C (columns) from f.
2. **Dimensions of B:** Read integers L_B (rows) and C_B (columns) of the structuring element.
3. **Structuring element:** Read matrix B with values 0 or 1, row by row.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-ep0404-erosao.html

Figure 4.33: EP04_04 Simulator: Planar Binary Erosion (g = f B)

4. **Data:** Read binary matrix f (values 0 or 1), row by row.
5. **Erosion:** Compute $e = f \ominus B$, using exactly the algorithm from EP04_04 (without reflecting B, without padding).
6. **Dilation:** Compute $g = e \oplus B$, using exactly the algorithm from EP04_03 (reflecting B, without padding)—but now applied to e, not to f.
7. **Output:** Display the resulting matrix g (the **opening** of f by B) with dimensions $L \times C$.

4.9.5.2 📌 Computational Constraints

- **Fixed order:** It is **always** erosion first, then dilation—the reverse order defines another operator (closing, from the next EP).
- **Same B:** The structuring element used in erosion and dilation must be identical.
- **No padding in either step.**

4.9.5.3 🧠 Theoretical Foundation

Concept	Meaning	Visual Impact
Anti-extensivity	$g \subseteq f$ always	The opening never creates a new pixel, only removes
Idempotence	opening(opening(f)) = opening(f)	Applying it again changes nothing further
Isolated points	Smaller than B	Are completely eliminated
Object core	Larger than B	Is recovered almost intact by the final dilation

4.9.5.4 📦 Input and Output Specification (VPL)

Input:

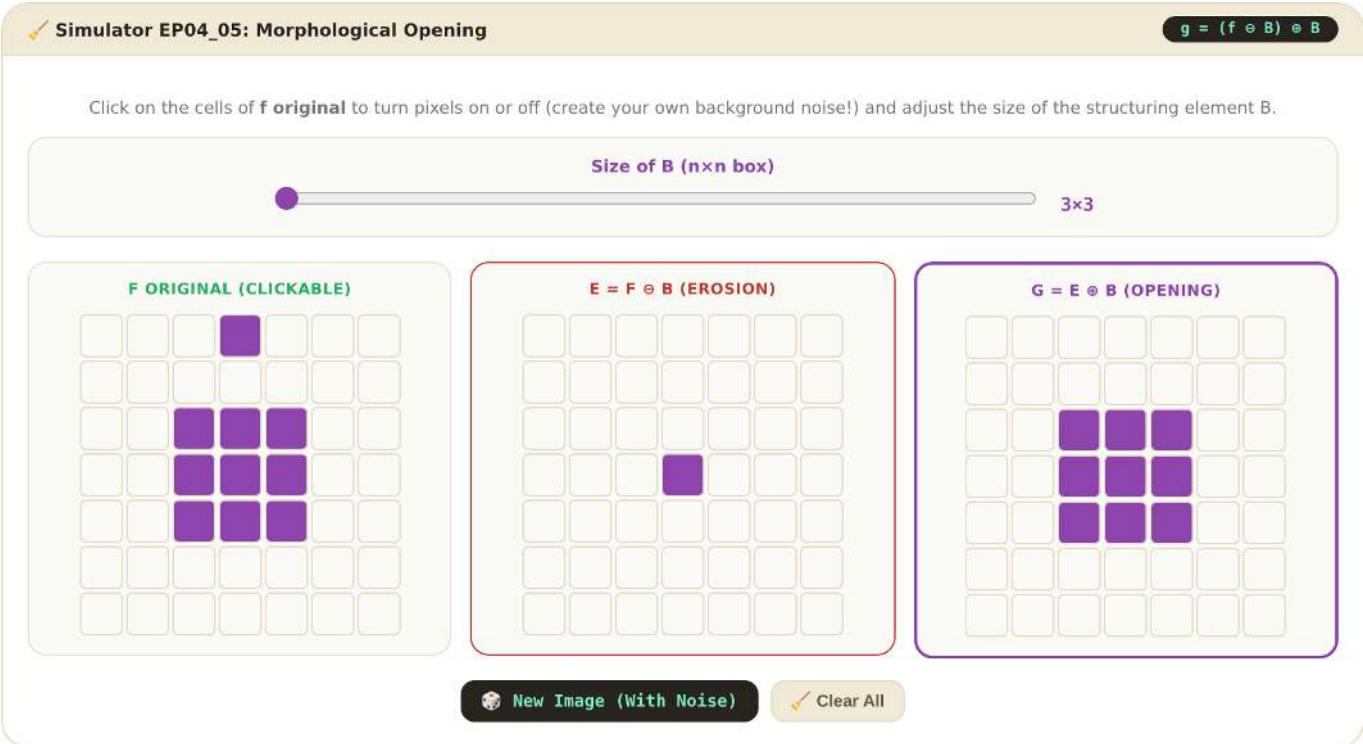
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer L_B.
- Line 4: Integer C_B.
- Next L_B lines: integer elements (0 or 1) of matrix B.
- Next L lines: integer elements (0 or 1) of matrix f.

Output:

- Resulting matrix with L rows and C columns, values 0 or 1.

4.9.5.5 📌 Examples

Input	Output	Observation
7	0 0 0 0 0 0 0	Isolated points and the thin protrusion
7	0 0 0 0 0 0 0	disappear; the central square survives
3	0 0 1 1 1 0 0	
3	0 0 1 1 1 0 0	
1 1 1	0 0 1 1 1 0 0	
1 1 1	0 0 0 0 0 0 0	
1 1 1	0 0 0 0 0 0 0	
0 0 0 0 0 0 0		
0 1 0 0 0 1 0		
0 0 1 1 1 0 0		
0 0 1 1 1 0 0		
0 0 1 1 1 1 0		
0 0 0 0 0 0 0		
0 1 0 0 0 0 1		



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-ep0405-abertura.html

Figure 4.34: EP04_05 Simulator: Morphological Opening (g = (f B) B)

```
%%writefile EP04_05.py
# Python Code
```

```
Overwriting EP04_05.py
```

```
TestSuite("EP04_05.py").run()
```

```
<IPython.core.display.HTML object>
```

4.9.6 EP04_06 🧩 Morphological Closing (Filling of Gaps)

In **fingerprint digitalization**, skin ridges sometimes become interrupted by dirt or dryness, creating small gaps in the continuous curve that should exist. **Closing** — dilation followed by erosion with the same structuring element — is the dual operator of opening: it **fills small holes and narrow indentations**, without significantly altering the external contour of the object. It is the standard step before extracting the skeleton of a fingerprint. See Figure 4.35 for a simulation of this EP.

4.9.6.1 📋 Implementation Guidelines

1. **Image dimensions:** Read the integers L (rows) and C (columns) from f.
2. **Dimensions of B:** Read the integers L_B (rows) and C_B (columns) of the structuring element.
3. **Structuring element:** Read the matrix B with values 0 or 1, row by row.
4. **Data:** Read the binary matrix f (values 0 or 1), row by row.
5. **Dilation:** Compute $d = f \oplus B$, using exactly the algorithm from EP04_03 (reflecting B, without padding).
6. **Erosion:** Compute $g = d \ominus B$, using exactly the algorithm from EP04_04 (without reflecting B, without padding) — now applied to d, not to f.
7. **Output:** Display the resulting matrix g (the **closing** of f by B) with dimensions $L \times C$.

4.9.6.2 📌 Computational Constraints

- **Fixed order:** It is **always** dilation first, then erosion — the reverse order corresponds to the opening from EP04_05.
- **Same B:** The structuring element used in the dilation and in the erosion must be identical.
- **No padding in either of the two stages.**

4.9.6.3 🧠 Theoretical Foundation

Concept	Meaning	Visual Impact
Extensivity	$g \supseteq f$ always	Closing never removes a pixel, only adds
Idempotence	$\text{close}(\text{close}(f)) = \text{close}(f)$	Applying it again changes nothing further
Small holes	Smaller than B	They are completely filled
Duality	$\text{close}(f) = \overline{\text{open}(\bar{f})}$	It is the opening applied to the "negative" of the image

4.9.6.4 📦 Input and Output Specification (VPL)

Input:

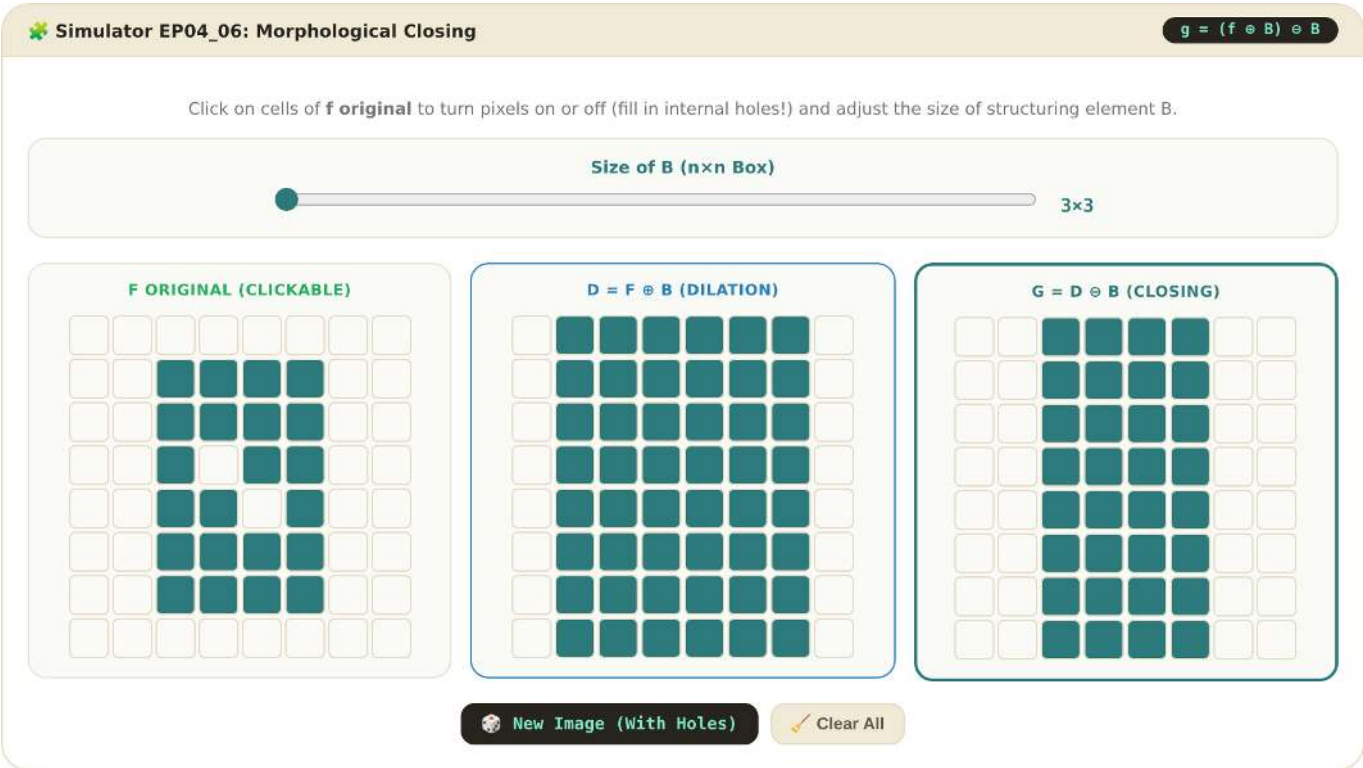
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer L_B.
- Line 4: Integer C_B.
- Next L_B lines: integer elements (0 or 1) of matrix B.
- Next L lines: integer elements (0 or 1) of matrix f.

Output:

- Resulting matrix in L rows and C columns, values 0 or 1.

4.9.6.5 📌 Examples

Input	Output	Observation
8 8 3 3 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0	0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0	The two non-adjacent internal holes are completely filled



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-ep0406-fechamento.html

Figure 4.35: EP04_06 Simulator: Morphological Closing (g = (f B) B)

```
%%writefile EP04_06.py
# Python code
```

```
Overwriting EP04_06.py
```

```
TestSuite("EP04_06.py").run()
```

```
<IPython.core.display.HTML object>
```

4.9.7 EP04_07 Weighted Dilation and Erosion (mm.dil1 / mm.ero1)

So far, the structuring element only indicated "this neighbor counts" or "does not count" — but in **digital elevation models** (used in GIS and urban drainage planning), each neighbor should have a **different weight** depending on the distance or the direction of the terrain. The **weighted** versions of dilation and erosion, implemented in `morph.py` as `mm.dil1(f, b)` and `mm.ero1(f, b)`, sum (or subtract) the weight of each neighbor before taking the maximum (or minimum) — generalizing everything done in the previous EPs. See Figure 4.36 for a simulation of this EP.

4.9.7.1 📋 Implementation Guidelines

1. **Image dimensions:** Read the integers L (rows) and C (columns) from f.
2. **Dimensions of b:** Read the integers L_B (rows) and C_B (columns) of the weighted structuring element.
3. **Weights:** Read the matrix b of **integer** weights (which may be negative, zero, or positive), row by row.
4. **Data:** Read the matrix f (the original image), row by row.
5. **Neighborhood without padding:** For each pixel (y, x), iterate over **all** positions (by, bx) of b (not only where it would equal 1 — here **every** weight participates), using the same offset as in previous EPs:
$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\tfrac{L_B}{2} + 0{,}5, \quad o_x = -\tfrac{C_B}{2} + 0{,}5$$
Discard every (v_y, v_x) outside $[0, L) \times [0, C)$.
6. **Weighted dilation:** Compute
$$g_{dil}(y, x) = \max\Big(f(y, x), \max_{(v_y, v_x)\ \text{valid}} \big(f(v_y, v_x) + b(by, bx)\big)\Big)$$
7. **Weighted erosion:** Compute, **using the same b and without reflection**:
$$g_{ero}(y, x) = \min\Big(f(y, x), \min_{(v_y, v_x)\ \text{valid}} \big(f(v_y, v_x) - b(by, bx)\big)\Big)$$
8. **Output:** Display **first** the complete matrix g_{dil}, and **then** the complete matrix g_{ero}.

4.9.7.2 📌 Computational Constraints

- **Neither reflects** b — the weighted version does not use reflection, even in dilation (unlike `mm.dil0`).
- **All weights participate:** There is no "$B = 1$" filter here; even weight 0 is included in the computation.
- **No padding:** neighbors outside the image are ignored, never virtually filled.
- **Type:** The output may contain negative values or values greater than 255 — there is **no** *clipping* in this EP.
- **Hint:** To remove overflow messages when exceeding uint8 limits, include at the beginning of the code:

```
import warnings
warnings.filterwarnings("ignore")
```

4.9.7.3 🧠 Theoretical Foundation

Concept	Meaning	Visual Impact
Positive weight	"Pulls" the neighbor's value upward in dilation	Simulates terrain rising in that direction
Negative weight	Reduces the neighbor's contribution	Simulates distance or directional attenuation
Weighted duality	$\text{ero1}(f, b) = -\text{dil1}(-f, b)$	The symmetry between the two operations is maintained even with weights

4.9.7.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer L_B.
- Line 4: Integer C_B.
- Next L_B lines: integer elements (may be negative) of the matrix b.
- Next L lines: integer elements of the matrix f.

Output:

- First, the matrix g_{dil} in L rows and C columns.
- Then, the matrix g_{ero} in L rows and C columns.

4.9.7.5 📌 Examples

Input	Output	Observation
3 3 3 3 0 1 0 1 2 1 0 1 0 10 20 30 40 50 60 70 80 90	50 60 61 80 90 91 81 91 92 8 9 19 9 10 20 39 40 50	Central weight 2 accelerates growth in dilation and shrinkage in erosion

```
%%writefile EP04_07.py
# Python code
```

```
Overwriting EP04_07.py
```

```
TestSuite("EP04_07.py").run()
```

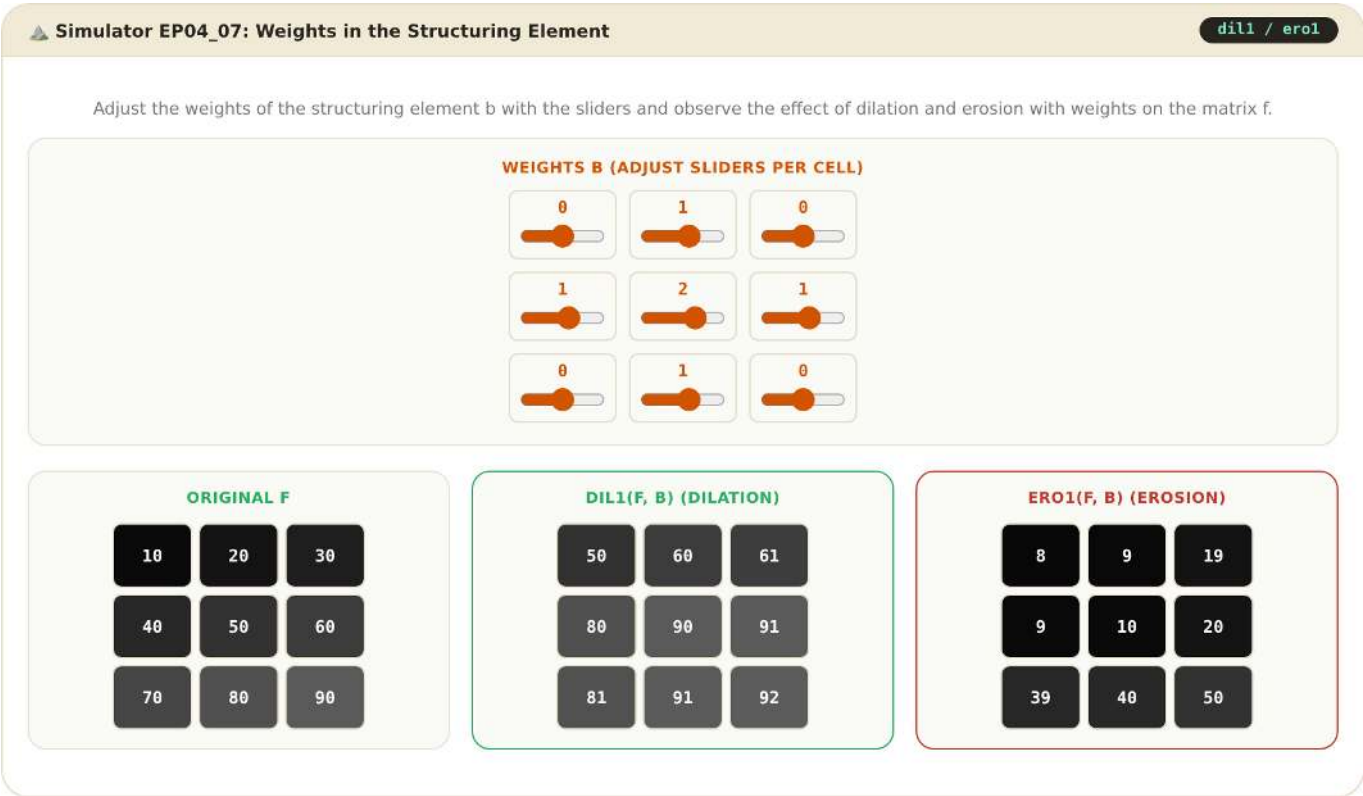
```
<IPython.core.display.HTML object>
```

4.9.8 EP04_08 🌋 Morphological Gradient, Top-hat, and Black-hat

In **automatic inspection of printed circuit boards**, three questions arise all the time: where are the **edges** of the components? Which **small bright details** (such as solder points) stand out from the background? What **dark recesses** (such as cracks) does the background conceal? A single erosion/dilation pair answers all three: the **morphological gradient** highlights contours, the **top-hat** reveals narrow peaks, and the **black-hat** reveals narrow valleys — three tools, one single neighborhood. See Figure 4.37 for a simulation of this EP.

4.9.8.1 📋 Implementation Guidelines

1. **Image dimensions:** Read the integers L (rows) and C (columns) from f.
2. **Dimensions of B:** Read the integers L_B (rows) and C_B (columns) of the structuring element.
3. **Structuring element:** Read the matrix B with values 0 or 1, row by row.
4. **Data:** Read the matrix f (the original image, in grayscale), row by row.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-ep0407-pesos.html

Figure 4.36: EP04_07 Simulator: Dilation and Erosion with Weights (mm.dil1 / mm.ero1)

5. **Basic operators:** Compute, exactly as in EPs 04_03 through 04_06:
   - $d = f \oplus B$ (dilation),
   - $e = f \ominus B$ (erosion),
   - opening $= e \oplus B$,
   - closing $= d \ominus B$.
6. **Morphological gradient:** $\text{grad}(y, x) = d(y, x) - e(y, x)$.
7. **Top-hat:** $\text{tophat}(y, x) = f(y, x) - \text{opening}(y, x)$.
8. **Black-hat:** $\text{blackhat}(y, x) = \text{closing}(y, x) - f(y, x)$.
9. **Output:** Display, **in this order**, the three complete matrices: gradient, top-hat, black-hat.

4.9.8.2 📌 Computational Constraints

- **No padding at any intermediate stage** — dilation, erosion, opening, and closing follow the same neighborhood rules as in the previous EPs.
- **No clipping:** the three outputs may contain any integer value (the gradient is always ≥ 0, but top-hat and black-hat may also be so).
- **Reuse:** d and e must be computed **only once** and reused to assemble opening, closing, and gradient.

4.9.8.3 🧠 Theoretical Foundation

Operator	Formula	What it reveals
Gradient	$d - e$	Edges: zero in flat regions, high at transitions
Top-hat	$f - \text{opening}(f)$	**Bright, thin** elements, smaller than B
Black-hat	$\text{closing}(f) - f$	**Dark, thin** elements, smaller than B

4.9.8.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer L_B.
- Line 4: Integer C_B.
- Next L_B lines: integer elements (0 or 1) of matrix B.
- Next L lines: integer elements of matrix f.

Output:

- Gradient matrix with L rows and C columns.
- Top-hat matrix with L rows and C columns.
- Black-hat matrix with L rows and C columns.

4.9.8.5 📌 Examples

Input	Output	Observation
9 9 3 3 1 1 1 1 1 1 1 1 1 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 80 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 2 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10	(gradient: 3×3 halo of 70 around $(2, 2)$ and 3×3 halo of 8 around $(6, 6)$, rest 0) (top-hat: single 70 at $(2, 2)$, rest 0) (black-hat: single 8 at $(6, 6)$, rest 0)	Isolated peak becomes top-hat; isolated valley becomes black-hat; both appear in the gradient

```
%%writefile EP04_08.py
# Python code
```

```
Overwriting EP04_08.py
```

```
TestSuite("EP04_08.py").run()
```

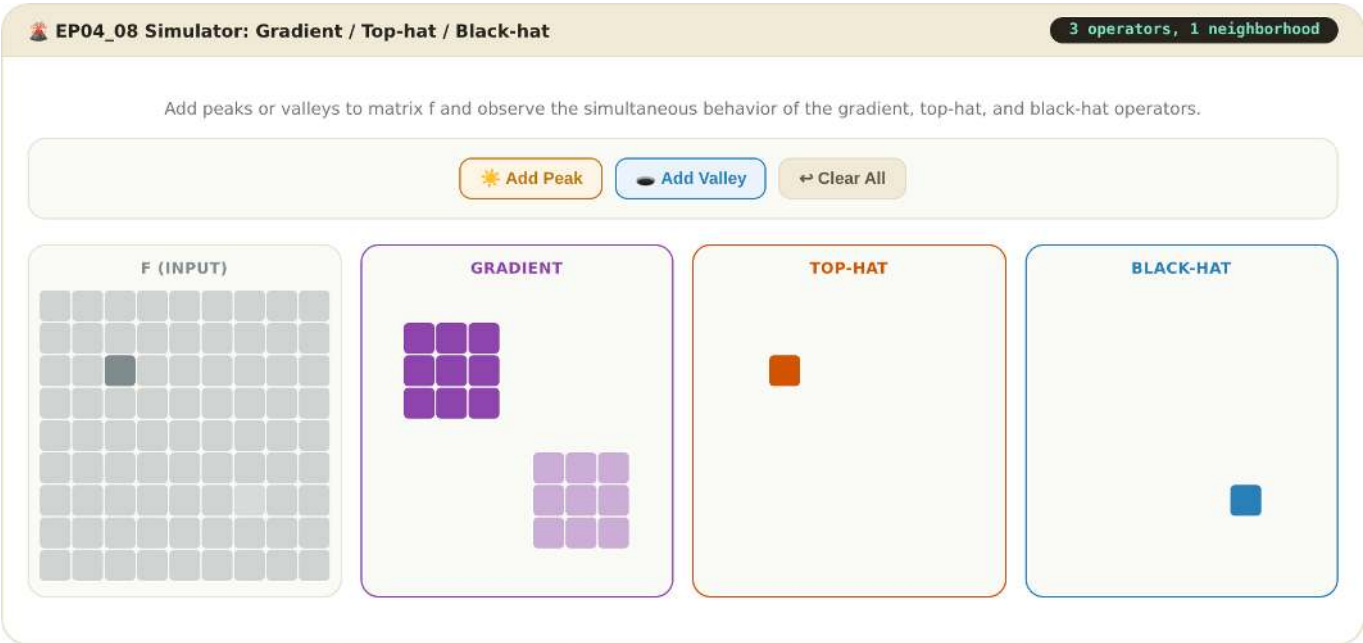
```
<IPython.core.display.HTML object>
```

4.9.9 EP04_09 Distance Transform and the Object's "Core"

In **mobile robotics**, when planning a route within a corridor, the robot wants to know not only *where* free space exists, but also **how far** each free point is from the nearest wall. The safest paths tend to pass through the corridor's "core," away from obstacles.

The **morphological distance transform** assigns to each pixel a value representing its distance to the nearest edge, according to the metric defined by the structuring element. Pixels near the edge receive low values, while more internal pixels receive higher values. The pixel with the maximum value corresponds to the most protected region of the object, often associated with its morphological center.

See Figure 4.38 for a simulation of this EP.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-ep0408-gradiente.html

Figure 4.37: EP04_08 Simulator: Morphological Gradient, Top-hat and Black-hat

4.9.9.1 📋 Implementation Guidelines

1. **Image dimensions:** read the integers L (rows) and C (columns) of image f.
2. **Dimensions of B:** read the integers L_B (rows) and C_B (columns) of the structuring element.
3. **Structuring element:** read the matrix b, containing value 0 at the center and negative values at other positions.
4. **Image:** read the binary matrix f (values 0 or 1), row by row.
5. **Preparation:** multiply the image by $L \times C$, ensuring that internal pixels have an initial value sufficiently high for distance propagation.
6. **Distance transform:** compute the distance matrix using the method `mm.dist1(f,b)`.
7. **Output:** display the matrix resulting from the distance transform.

4.9.9.2 📌 Computational Constraints

- Use the weighted erosion implementation provided by the library.
- The structuring element may contain arbitrary negative values.
- The transform must be obtained by iteratively applying weighted erosions until a fixed point is reached.

⚠️ **Crucial Note on Matrix Reading:** Since the structuring element may contain negative integer values (e.g., `-1` and `-99`), **do not use the `mm.readImg` function to read matrix** b. This function converts data to `uint8` type, causing *underflow* and corrupting negative values. Read the L_B rows of b manually using the default `int` type. Image f can continue to be read normally using `mm.readImg`.

4.9.9.3 🧠 Theoretical Foundation

Concept	Meaning	Visual Impact
dist(y, x)	Morphological distance to the nearest edge according to the metric defined by b	More internal pixels receive higher values
Maximum value	Pixel farthest from the edge	Approximates the object's morphological center
Weighted structuring element	Defines the displacement costs between neighboring pixels	Determines the distance metric used

Concept	Meaning	Visual Impact
Thin objects	Narrow regions of the object	Produce low distance values

4.9.9.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: integer L.
- Line 2: integer C.
- Line 3: integer L_B.
- Line 4: integer C_B.
- Next L_B lines: integer elements of matrix b.
- Next L lines: binary elements (0 or 1) of matrix f.

⚠️ **Implementation note:** The elements of matrix f (0 or 1) must be multiplied by **255** to generate an adequate binary image (0 and 255) before applying the Distance Transform (DT).

Output:

- Distance transform matrix with L rows and C columns.

4.9.9.5 📌 Example

Input	Output	Observation
5 9 3 3 -99 -1 -99 -1 0 -1 -99 -1 -99 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 2 2 2 2 2 1 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0	Result of the distance transform.

Note: the value `-99` acts as a practical approximation of $-\infty$, preventing propagation along diagonals. Thus, only horizontal and vertical neighbors contribute to the distance, producing the Manhattan distance.

```
%%writefile EP04_09.py
# Python code
```

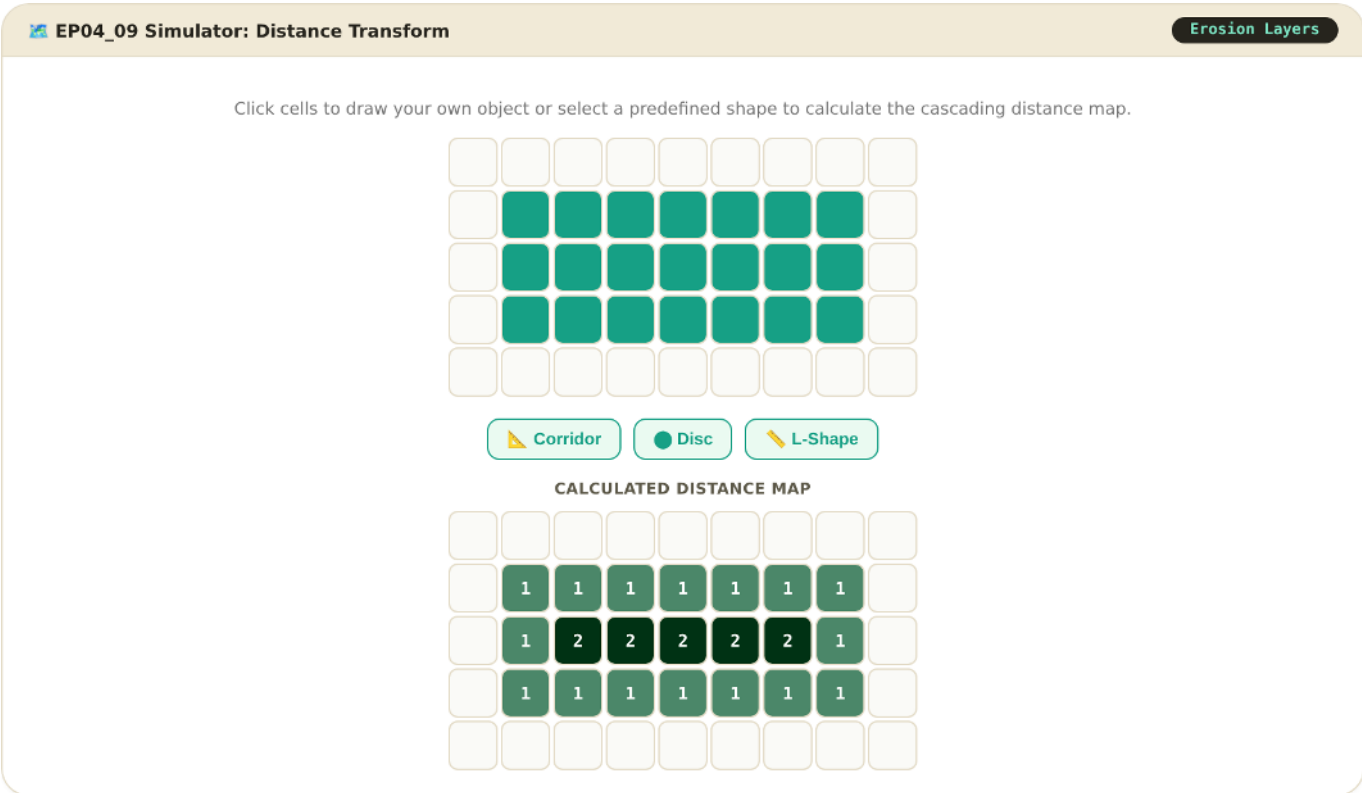
```
Overwriting EP04_09.py
```

```
TestSuite("EP04_09.py").run()
```

```
<IPython.core.display.HTML object>
```

4.9.10 EP04_10 🪙 *Blob* Separation, Labeling, and Descriptors

In a **coin production line**, it is common for pieces to touch one another on the conveyor belt, forming a single connected blob in the image—a naive count would yield the wrong total. The classic solution combines



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-ep0409-distancia.html

Figure 4.38: Simulador EP04_09: Distância por Transformada (Camadas de Erosão)

morphological operations and connectivity analysis: first, an **erosion** reduces or breaks fragile connections between objects, and then **connected component labeling** separates each object into a distinct region. Finally, **geometric descriptors** (area and bounding box) summarize each detected component.

See Figure 4.39 for a simulation of this EP.

4.9.10.1 📋 Implementation Guidelines

1. **Image dimensions:** read the integers L (rows) and C (columns) from f.
2. **Dimensions of B:** read the integers L_B (rows) and C_B (columns) of the structuring element.
3. **Structuring element:** read the matrix B, containing values 0 or 1, row by row.
4. **Data:** read the binary matrix f (values 0 or 1), row by row.
5. **Separation:** compute
$$f_{ero} = f \ominus B$$
using flat binary erosion (as in EP04_04), eliminating fragile connections between objects.
6. **Labeling:** on f_{ero}, identify connected components using connectivity defined by the neighborhood B. Labeling must follow *raster* scanning: upon finding an unlabeled pixel with value 1, assign a new increasing integer label starting from 1 and propagate that label to the entire connected region.
7. **Descriptors:** for each label k, compute:
   - **Area:** number of pixels belonging to the label;
   - **Bounding box:**
$$(y_{min}, x_{min}, y_{max}, x_{max})$$
8. **Output:** display the total number of labels and then one line per label in the format:
$$k,\ \text{area},\ y_{min},\ x_{min},\ y_{max},\ x_{max}$$

4.9.10.2 📌 Computational Constraints

- Erosion must be applied before labeling.
- Connectivity is fixed and defined by the neighborhood above.
- The structuring element B does not interfere with labeling connectivity.
- No padding in any step.
- The order of labels follows first discovery in *raster* scanning.

4.9.10.3 🧠 Theoretical Background

Concept	Meaning	Impact
Thin bridge	Narrow connection between objects	Can be removed by morphological erosion
Connectivity	Defined by the set $\mathcal{N}(y, x)$	Determines which pixels belong to the same component
Area	Number of pixels per component	Direct estimate of object size
Bounding box	Spatial extent of the label	Geometric summary of the component

4.9.10.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: integer L
- Line 2: integer C
- Line 3: integer L_B
- Line 4: integer C_B
- Next L_B lines: matrix B
- Next L lines: matrix f

Output:

- Line 1: total number of labels found
- Following lines:

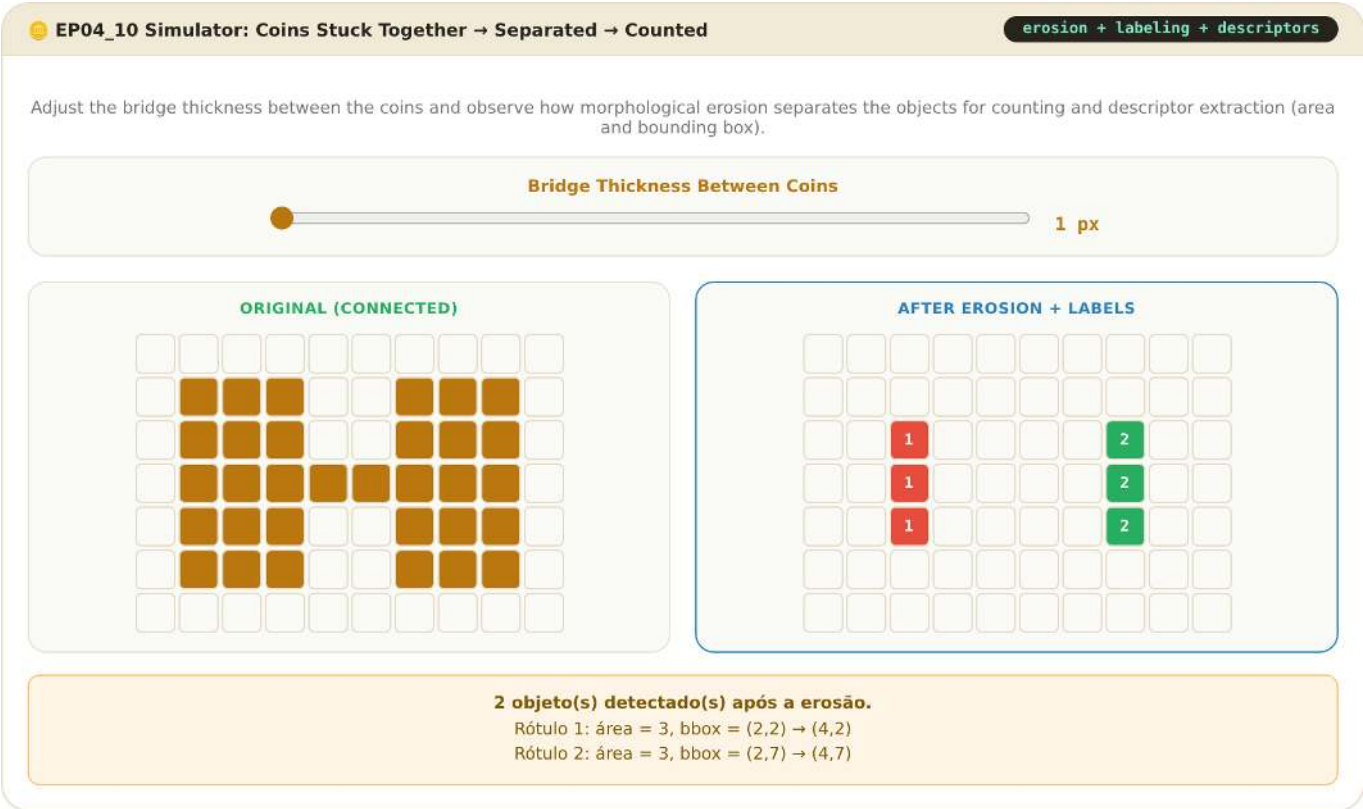
$$k,\ \text{area},\ y_{min},\ x_{min},\ y_{max},\ x_{max}$$

```
%%writefile EP04_10.py
# Python code
```

```
Overwriting EP04_10.py
```

```
TestSuite("EP04_10.py").run()
```

```
<IPython.core.display.HTML object>
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap04/sim-ep0410-rotulacao.html

Figure 4.39: Simulador EP04_10: Blob Separation, Labeling, and Descriptors

Chapter 5

Transforms and Compression

cap05/images/cabeca.jpg

In the previous chapters, all operations were performed **in the spatial domain**, where algorithms act directly on pixel intensity values.

This chapter presents a complementary approach: the **frequency domain**, in which the image is represented by the spatial variations of intensity, rather than only by the individual pixel values.

The concept of **spatial frequency** describes how quickly intensity varies across the image. Slow variations correspond to **low frequencies**, while edges, fine details, and noise correspond to **high frequencies**.

This representation relies on the fact that any discrete digital image can be decomposed into a combination of **orthogonal functions**. The **Fourier Transform** uses a **basis of two-dimensional complex exponentials** (equivalent to sinusoids with specific orientation and frequency). Other transforms, such as the **Cosine Transform (DCT)** and the **Wavelet Transform (DWT)**, use different families of basis functions — two-dimensional cosines in the case of the DCT, and functions with compact support in the case of wavelets.

Among the main applications of this representation are:

1. **Frequency-domain filtering**, to attenuate or enhance certain frequency bands;
2. **Multiresolution analysis through wavelet transforms**, which represents structures at different scales;
3. **Image compression**, by reducing the number of coefficients needed to represent the image.

5.1 Objectives

By the end of this chapter, you will be able to:

- **Interpret the Fourier spectrum** of an image, distinguishing magnitude, phase, and frequency components;
- **Apply the Convolution Theorem** to perform frequency-domain filtering using the Fast Fourier Transform (FFT);
- **Design and analyze frequency-domain filters**, understanding the operation of low-pass, high-pass, and notch filters;
- **Understand multiresolution analysis via wavelet transforms** and its application to the hierarchical representation of images;
- **Describe the image compression process**, including the Discrete Cosine Transform (DCT) and coefficient quantization;
- **Select image storage formats**, such as JPEG, PNG, and WebP, according to application requirements.

5.2 Environment Setup

```
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup()
from morph import mm
import numpy as np
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0
```

5.3 2D Discrete Fourier Transform

Fourier analysis is based on the principle that any periodic signal can be represented as a sum of sinusoidal functions with different frequencies, amplitudes, and phases. This concept also applies to digital images, allowing them to be represented in the **frequency domain** instead of the spatial domain.

Figure 5.1 illustrates this decomposition for a one-dimensional signal. In the case of an image, the Discrete Fourier Transform (DFT) converts the intensity matrix $f(x, y)$ into a set of coefficients that describes the contribution of the different spatial frequencies present in the image.

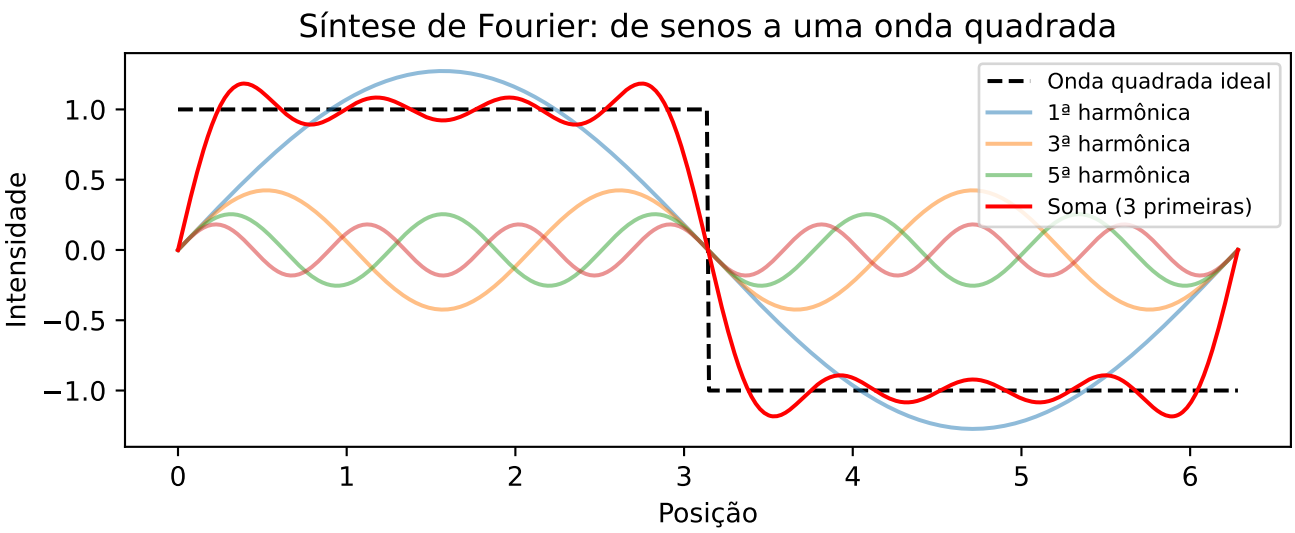


Figure 5.1: 1D Fourier decomposition: a square wave (dashed line) is approximated by the sum of the first sinusoids (colored lines). The more terms, the better the approximation.

5.3.1 Simulator: Reconstructing Signals with Sinusoids

Before studying two-dimensional images, the simulator in Figure 5.2 illustrates the principle of Fourier analysis for one-dimensional signals: **a waveform can be approximated by the sum of sinusoids with different frequencies and amplitudes**.

As new terms are added, the sum of the sinusoids (black curve) approaches the reference waveform (dashed). The lower plot displays the amplitude spectrum, indicating the contribution of each frequency to the reconstruction of the signal.

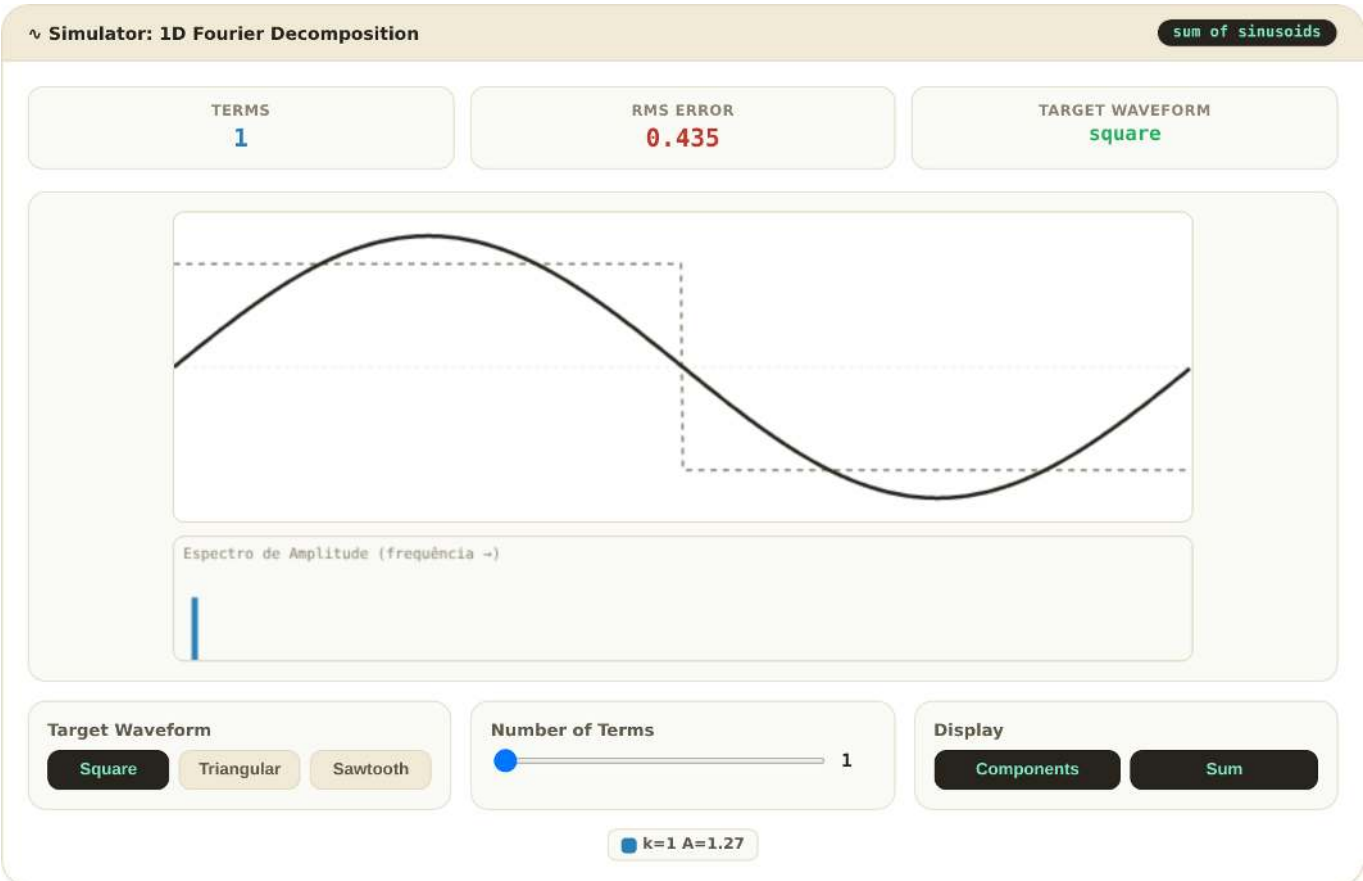
Activity

Explore the simulator and answer:

1. How many terms are needed to obtain a good approximation of the square wave?
2. Which of the three waveforms converges most rapidly? Justify your answer.
3. How does the amplitude spectrum change when switching from the square wave to the triangular wave?

Answers

1. How many terms are needed for a good approximation of the square wave?
With approximately 15 to 20 terms, the waveform already closely approximates the reference. However, near the discontinuities, a small oscillation remains, known as the **Gibbs phenomenon**, which does not disappear even with the addition of more terms.
2. Which waveform converges most rapidly? Why?
The **triangular wave** converges most rapidly, because the amplitudes of its harmonics decay faster than those of the square wave and the sawtooth wave. As a result, few terms already produce a good approximation.
3. How does the spectrum change between the square wave and the triangular wave?
Both contain only **odd harmonics**, but in the triangular wave, the amplitudes decrease much more rapidly. Thus, few harmonics are sufficient to reconstruct the signal with good accuracy.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap05/sim-05-freq.html

Figure 5.2: Interactive simulator of 1D Fourier decomposition: visualization of the sum of sinusoids with different frequencies, amplitudes and phases. Add terms and observe convergence to arbitrary waveforms.

5.3.2 Interpretation of the Frequency Spectrum

When applying the Discrete Fourier Transform (DFT) to an image and visualizing the magnitude of its coefficients (see Figure 5.5), one obtains the **magnitude spectrum**, which shows the distribution of spatial frequencies present in the image.

The coefficient located at the origin of the DFT, known as the **DC component** (*Direct Current*), corresponds to the zero frequency and represents the average intensity of the image. By convention, this coefficient is stored in the upper left corner of the spectrum. To facilitate its interpretation, the **FFT Shift** operation is applied, which shifts the DC component to the center of the image. After this shift, low frequencies are concentrated in the central region, while high frequencies are near the edges, as summarized in Table 5.1.

Table 5.1: Correspondence between the regions of the magnitude spectrum after applying the FFT Shift.

Spectrum region	Predominant components	Examples in the image
Center (low frequencies)	Slow spatial variations	Illumination, homogeneous regions, and global shapes
Intermediate region (mid frequencies)	Intermediate-scale variations	Textures and repetitive patterns
Edges (high frequencies)	Rapid spatial variations	Contours, fine details, and noise

This organization facilitates the interpretation of the spectrum and the design of filters. Attenuating low frequencies reduces global intensity variations, while attenuating high frequencies smooths the image by reducing fine details and part of the noise.

5.3.3 The Grid Experiment: Building an Image from a Single Coefficient

Before presenting the mathematical formulation of the Discrete Fourier Transform (DFT), it is useful to analyze its inverse, called the Inverse Discrete Fourier Transform (IDFT). Consider a spectrum in which all coefficients are zero, except one. An example of this construction is presented in the code of Figure 5.3 and can be explored interactively in the simulator of Figure 5.4.

The reconstructed image is a two-dimensional sinusoid. The position of the coefficient in the spectrum determines its **orientation** and its **spatial frequency**, while its magnitude and phase define, respectively, its amplitude and spatial displacement. Thus, each DFT coefficient represents a sinusoidal component, and the original image can be reconstructed by summing all these components.

```
import cv2

N_grid = 100
espectro_vazio = np.zeros((N_grid, N_grid), dtype=complex)

# Turning on a single point (frequency) off-center
u0, v0 = 10, 5
espectro_vazio[N_grid//2 - v0, N_grid//2 - u0] = 1000

# Returning to the spatial domain (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)

# Visual highlight of the point
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=["Spectrum (1 active point)", "Resulting 2D Wave (IDFT)"],
        cols=2, figsize=(10, 4))
```

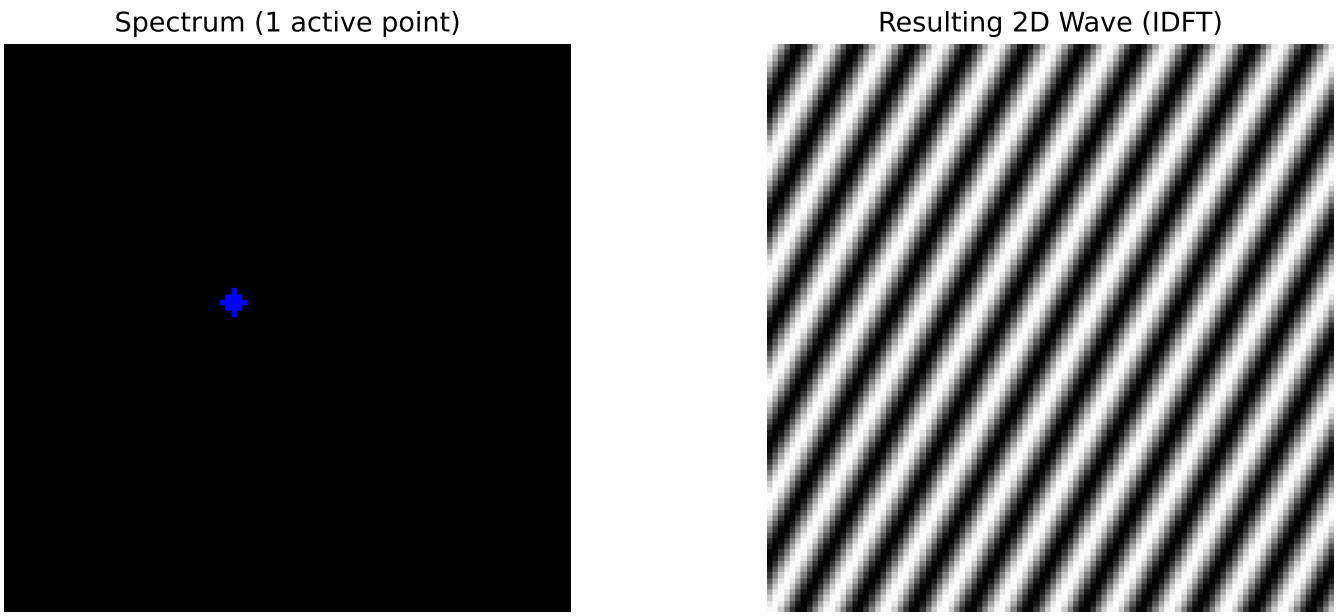
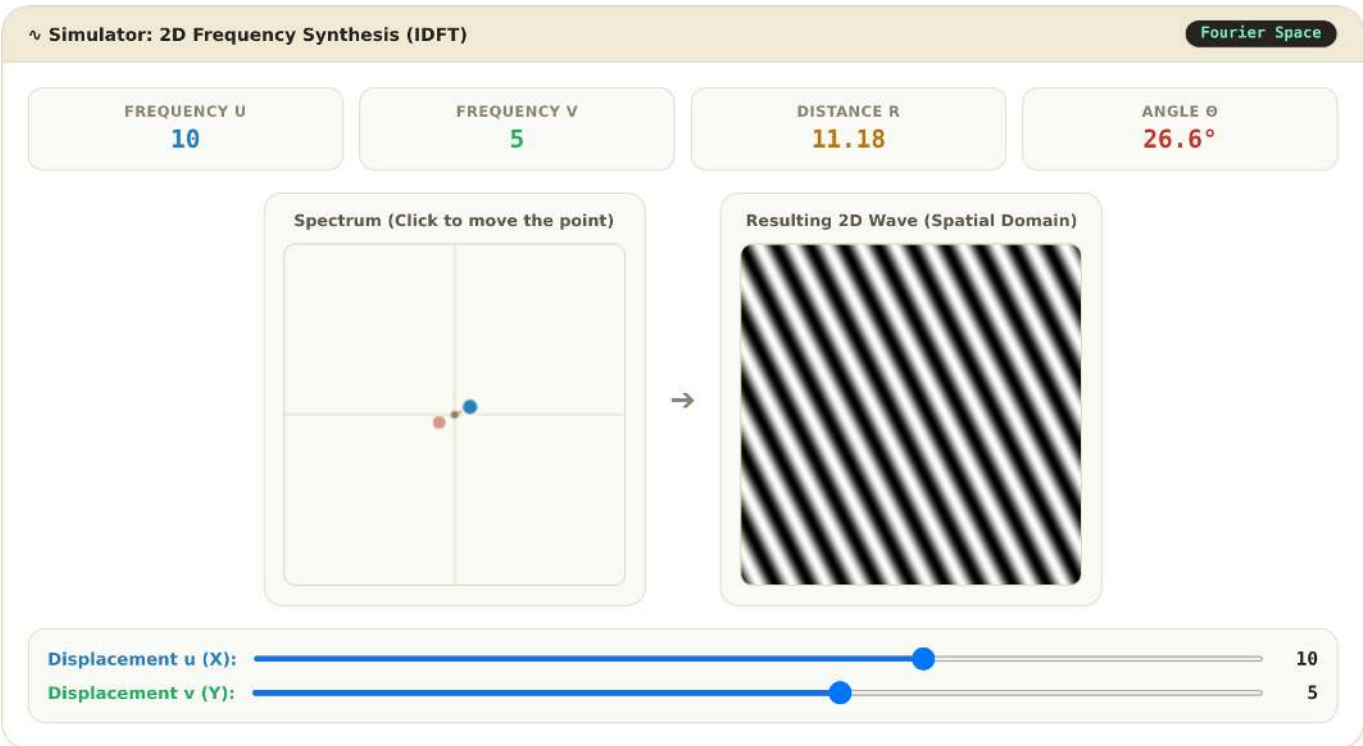


Figure 5.3: Every frequency in the spectrum (isolated point) corresponds to a rotated 2D sine wave in the spatial domain.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap05/sim-05-grade-2d.html

Figure 5.4: Interactive simulator of 2D Fourier synthesis. Change the horizontal (u) and vertical (v) position of the coefficient in the centered frequency spectrum and observe how the distance from the center dictates the spatial frequency (thickness) and the angle dictates the orientation of the generated sine wave.

5.3.4 Mathematical Definition

Consider an image $f(x, y)$ with dimensions $M \times N$. Its **2D Discrete Fourier Transform** (DFT) is defined by:

$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y)\, e^{-j2\pi\left(\frac{ux}{M} + \frac{vy}{N}\right)} \tag{5.1}$$

where $u = 0, 1, \ldots, M-1$ and $v = 0, 1, \ldots, N-1$ represent the discrete frequencies in the horizontal and vertical directions, respectively. The exponential term corresponds to a two-dimensional sinusoid, whose frequency and orientation are determined by the indices (u, v).

The **2D Inverse Discrete Fourier Transform** (IDFT) reconstructs the original image from its coefficients:

$$f(x, y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v)\, e^{j2\pi\left(\frac{ux}{M} + \frac{vy}{N}\right)} \tag{5.2}$$

Equations Equation 5.1 and Equation 5.2 show that the DFT and the IDFT form a pair of transformations: the former converts the image into the frequency domain, while the latter exactly reconstructs the original image from its coefficients.

> **On the symbol j**
>
> The term j denotes the **imaginary unit**, defined by $j^2 = -1$. In engineering and signal processing, j is adopted instead of i to avoid conflict with the notation for electric current. Its use in the complex exponential, governed by Euler's formula ($e^{j\theta} = \cos\theta + j\sin\theta$), allows for a compact representation of the amplitude and phase of each spatial frequency present in the image.

> **What is the DC component?**
>
> The coefficient $F(0, 0)$, termed the **DC component** (*Direct Current*), equals the sum of the intensities of all pixels in the image (see Figure 5.5):
>
> $$F(0, 0) = MN\,\bar{f},$$
>
> where $\bar{f}$ is the average intensity of the image. Therefore, the DC component represents the mean intensity level and, in most natural images, holds the largest magnitude of the spectrum.
> The remaining coefficients represent variations around this mean. After applying the **FFT Shift**, the DC component is moved to the center of the spectrum, concentrating low frequencies in the central region and high frequencies at the edges.

5.3.5 Magnitude and Phase

Each coefficient of the Discrete Fourier Transform (DFT) is a complex number and can be written as

$$F(u, v) = R(u, v) + j\, I(u, v),$$

where $R(u, v)$ and $I(u, v)$ correspond, respectively, to the **real** and **imaginary** parts of the coefficient. From Equation Equation 5.1, we obtain

$$R(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) \cos\left(2\pi \left(\frac{ux}{M} + \frac{vy}{N}\right)\right),$$

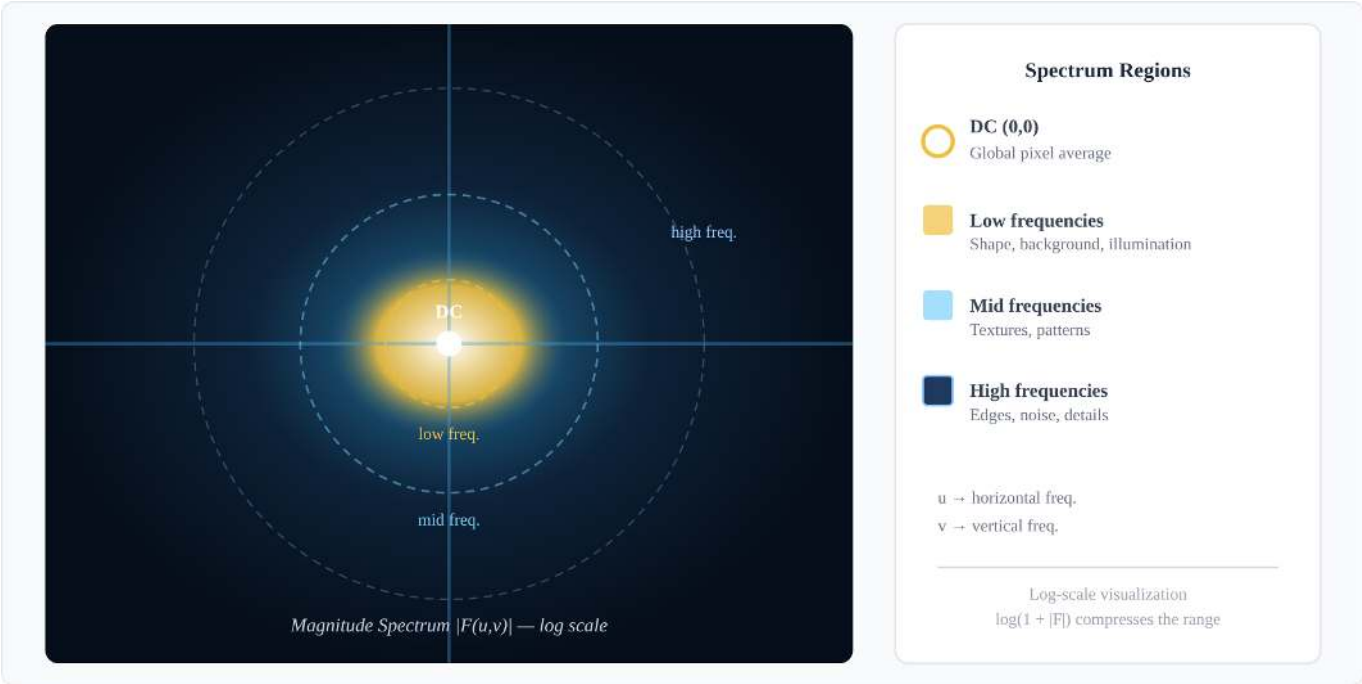


Figure 5.5: Conceptual diagram of the centered 2D Fourier spectrum.

and

$$I(u,v) = -\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)\sin\left(2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)\right).$$

From this representation, two fundamental quantities are defined:

- **Magnitude**, which indicates the intensity of the frequency component,

$$|F(u,v)| = \sqrt{R(u,v)^2 + I(u,v)^2};$$

- **Phase**, which determines the spatial alignment (or displacement) of the component,

$$\phi(u,v) = \operatorname{atan2}(I(u,v),\ R(u,v)).$$

Thus, each coefficient can also be written in its polar form,

$$F(u,v) = |F(u,v)|\, e^{j\phi(u,v)}.$$

The Fourier spectrum can, therefore, be visualized by means of two distinct images: the **magnitude spectrum**, typically used to analyze the distribution of frequencies, and the **phase spectrum**, which describes the spatial organization of the sinusoidal components.

Although the magnitude spectrum is the most used for visual inspection, the phase contains a large portion of the structural information of the image. The combination of magnitude and phase allows the exact reconstruction of the original image via the IDFT.

5.3.6 What Do Magnitude and Phase Carry?

A classic demonstration consists of combining the magnitude of one image with the phase of another and reconstructing the result. This experiment highlights that:

- **Phase** preserves the spatial structure of the image, including the position of objects, their contours, and their geometry. Small changes in phase can lead to major visual differences.
- **Magnitude** controls how energy is distributed among spatial frequencies, primarily influencing contrast and texture.

When an image is reconstructed with the magnitude of A and the phase of B, the result tends to **resemble B more than A**, showing that phase is the main component responsible for the spatial organization of the scene. However, magnitude remains important, as it modulates the contrast of the reconstructed structures. Thus, a faithful reconstruction depends on the consistent combination of magnitude and phase.

An example of this behavior is presented in Figure 5.6.

Analogy with Audio: Limitations and Caveats

The phase of a signal plays distinct roles in audio and images:

- **Stereo or multichannel audio:** the relative phase between channels is fundamental for perceiving the position of sound sources, through interaural time differences (ITD).
- **Monaural audio:** absolute phase has little direct perceptual influence.
- **Images (DFT):** phase is the main factor responsible for the spatial organization of the scene, while magnitude modulates contrast and the distribution of energy across frequencies.

In both domains, **magnitude** is related to the intensity of frequency components: in audio, it influences timbre and perceived loudness; in images, it influences contrast and texture.

```
#   Experiment: The Importance of Phase
#   Image Loading
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))

# Create a synthetic B image (geometric pattern)
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)

# Phase Swap
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=["Image A", "Image B", "Mag(A) + Phase(B)", "Mag(B) + Phase(A)"],
    cols=4, figsize=(16, 4)
)

print("  Phase preserves edges and contours; magnitude controls contrast and")
```

```
print("texture. In stereo audio, phase affects spatial location; in")
print("images, it determines the organization of the scene.")
```

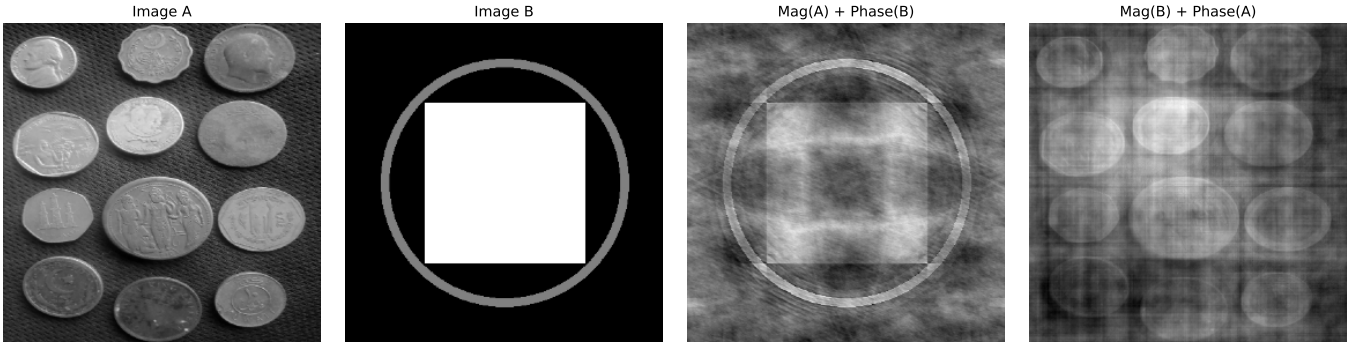


Figure 5.6: Phase swap experiment: Image A (coins) and Image B (geometric pattern) reconstructed with swapped magnitudes and phases. The result shows that the visual structure is **much more sensitive to phase** than to magnitude: when B's phase is preserved, the resulting image retains B's spatial organization, even with A's magnitude. Magnitude, in turn, mainly influences contrast and texture. Note that reconstruction quality is not perfect — visible artifacts exist —, highlighting the interdependence between phase and magnitude for a faithful image representation.

```
 Phase preserves edges and contours; magnitude controls contrast and
texture. In stereo audio, phase affects spatial location; in
images, it determines the organization of the scene.
```

5.4 Convolution Theorem and Filtering Strategies

The **Convolution Theorem** establishes a fundamental relationship between the spatial and frequency domains:

$$f(x,y) \circledast h(x,y) \overset{\mathcal{F}}{\longleftrightarrow} F(u,v)\,H(u,v) \tag{5.3}$$

where $\circledast$ denotes **discrete circular convolution**. Thus, the convolution between an image $f(x,y)$ and a filter $h(x,y)$ can be replaced by the multiplication of their spectra.

In practice, to obtain the same result as the linear convolution performed in the spatial domain, **zero-padding** is applied before the Fast Fourier Transform (FFT), avoiding artifacts at the image borders.

However, frequency-domain filtering is not always the most efficient alternative. For filters such as the Gaussian and the Box Filter, the **separability** property allows a significant reduction in the computational cost of convolution in the spatial domain.

5.4.1 Separable vs. Non-Separable *Kernel*

A ***separable kernel*** can be written as the outer product of two one-dimensional vectors,

$$H = v\,h^T,$$

allowing the two-dimensional convolution to be replaced by two consecutive one-dimensional convolutions: one in the horizontal direction and another in the vertical direction.

In contrast, a ***non-separable kernel*** does not admit such a decomposition, and therefore its convolution must be performed directly over the two-dimensional neighborhood.

In practice, for a kernel of dimension $K \times K$, direct convolution requires K^2 multiplications per pixel, whereas a separable kernel requires only $2K$ multiplications, significantly reducing the computational cost.

5.4.2 Computational Efficiency Analysis

Consider an image of dimensions $M \times N$ and a square filter of size $K \times K$. Table 5.2 compares the complexity of the main filtering strategies.

Table 5.2: Comparison of the complexity of direct convolution, separable convolution, and filtering via the Fast Fourier Transform (FFT).

Filtering Method	Asymptotic Complexity	Dependence on K	Typical Application
Non-separable spatial	$\mathcal{O}(MNK^2)$	Quadratic	Small, non-separable *kernels*
Separable spatial	$\mathcal{O}(MNK)$	Linear	Gaussian and mean filters
Via FFT	$\mathcal{O}(MN\log(MN))$	Independent of K	Large *kernels*

For small *kernels*, spatial convolution, especially when the filter is separable, tends to be more efficient due to the low cost of operations. As the *kernel* size increases, FFT-based filtering becomes more advantageous, since its cost is practically independent of the filter dimension.

5.4.3 Discussion of experimental results

The graph obtained in the test with the coin image (2560×1920), presented in Figure 5.7, confirms the behavior predicted by the computational complexity analysis.

1. **Non-separable convolution ($\mathcal{O}(MNK^2)$)**
   Direct convolution exhibits quadratic growth with kernel size. For small values of K, the cost is low, but it increases rapidly as the kernel grows, becoming unfeasible for real-time applications.

2. **FFT-based filtering ($\mathcal{O}(MN\log(MN))$)**
   The cost of the FFT depends only on the image size, being independent of K. Therefore, its performance remains approximately constant as the kernel varies, making it advantageous for large or non-separable filters.

3. **Separable convolution ($\mathcal{O}(MNK)$)**
   Decomposing the kernel into two one-dimensional filters significantly reduces the computational cost. In practice, this approach tends to be the most efficient for separable filters, especially in optimized implementations.

In general, the choice of method depends on the size and structure of the kernel. Separable filters are more efficient in the spatial domain, while the FFT becomes more advantageous for large kernels or multiple convolutions in the frequency domain.

$$g = \mathcal{F}^{-1}[\mathcal{F}(f) \cdot \mathcal{F}(h)] \quad \text{(FFT)} \qquad g = f \circledast h \quad \text{(direct convolution)} \qquad g = (f \circledast v) \circledast h^T \quad \text{(separable)} \tag{5.4}$$

where:

- $f(x, y)$ represents the input image;
- $h(x, y)$ is the two-dimensional filter kernel;
- v and h^T are, respectively, the vertical and horizontal vectors that compose the separable kernel.

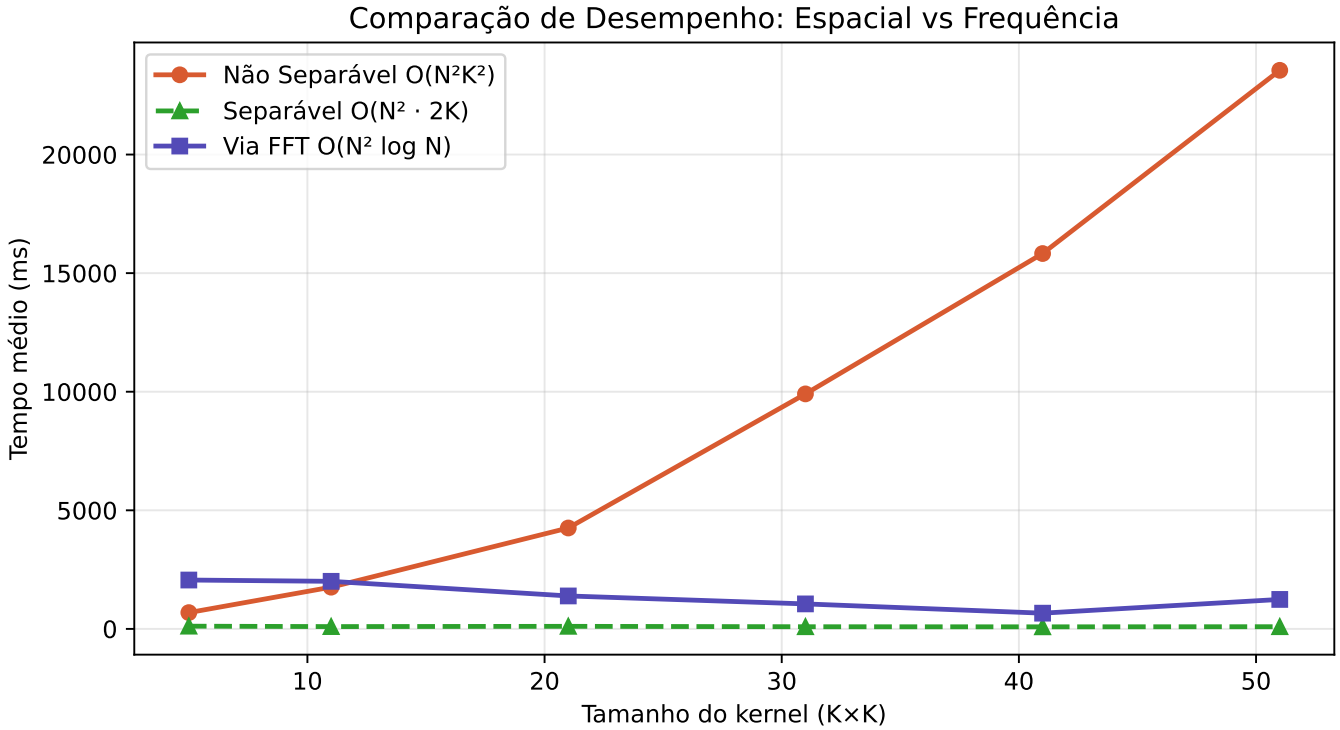


Figure 5.7: Efficiency comparison: Non-Separable Convolution (Spatial 2D), Separable (Spatial 1D) and via FFT.

! The Problem of Circular Convolution (*Wrap-Around*)

The Discrete Fourier Transform (DFT) assumes that the image is **periodically extended in space**, meaning that its borders repeat indefinitely.
Under this condition, multiplication in the frequency domain corresponds to a **circular convolution** in the spatial domain. Consequently, opposite regions of the image (top and bottom, left and right) begin to interact artificially, as illustrated in Figure 5.8.
Applying zero-padding before the FFT reduces this effect by extending the image with null values at the borders, approximating the result of linear convolution. This behavior can be interpreted in light of the Convolution Theorem, presented in Figure 5.9.

```
#   Define M and N
M, N = img_gray.shape  # added line

# Simulation of a brutal shift filter
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*120/M + v*120/N))

# Filtering WITHOUT padding (causes 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", "Filtering w/o Padding (Leakage)"], cols=2, figsize=(10, 4))
```

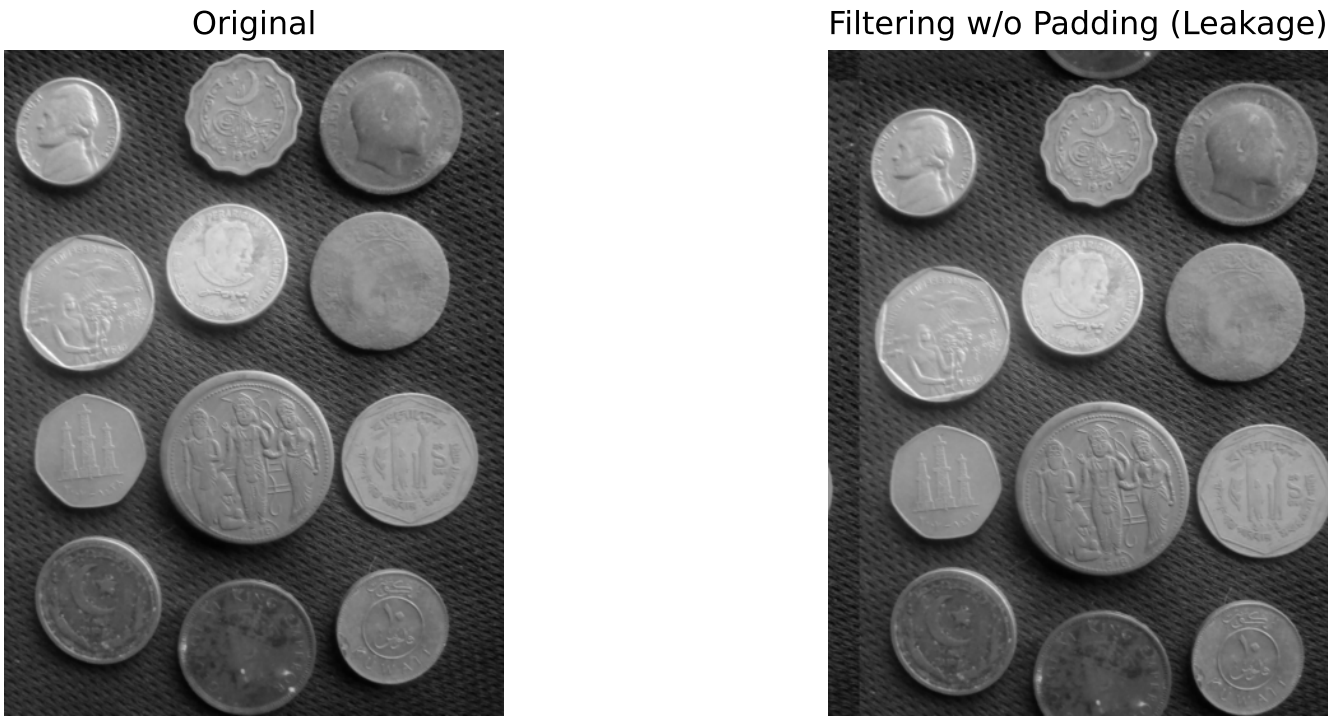


Figure 5.8: Without *padding*, a severe shift makes the image leak to the opposite side (circular convolution).

```
#   Gaussian Kernel 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)    # separable 2D kernel

#   Method 1: Direct spatial convolution
f_float  = img_gray.astype(np.float64)
conv_esp = cv2.filter2D(f_float, -1, kernel, borderType=cv2.BORDER_CONSTANT)

#   Method 2: Frequency multiplication (via FFT)
M, N     = f_float.shape
# Padding for linear convolution (avoids circular aliasing)
Mpad     = 2 ** int(np.ceil(np.log2(M + K - 1)))
Npad     = 2 ** int(np.ceil(np.log2(N + K - 1)))

# Positions the kernel with the origin at (0,0) and zero padding
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))

# Crop to compensate for the shift introduced by kernel positioning
offset   = K // 2
conv_freq_crop = conv_freq[offset:offset+M, offset:offset+N]

#   Numerical verification
diff = np.abs(conv_esp - conv_freq_crop)
print(f"Maximum difference  (|conv_spatial - conv_freq|): {diff.max():.2e}")
```

```
print(f"Mean difference   (|conv_spatial - conv_freq|): {diff.mean():.2e}")
print(f"→ Convolution Theorem numerically verified.")

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",
        "Spatial convolution",
        "Frequency multiplication",
        f"Difference (max={diff.max():.1e})"
    ],
    cols=4, figsize=(16, 5)
)
```

```
Maximum difference  (|conv_spatial - conv_freq|): 2.56e-13
Mean difference   (|conv_spatial - conv_freq|): 2.76e-14
→ Convolution Theorem numerically verified.
```

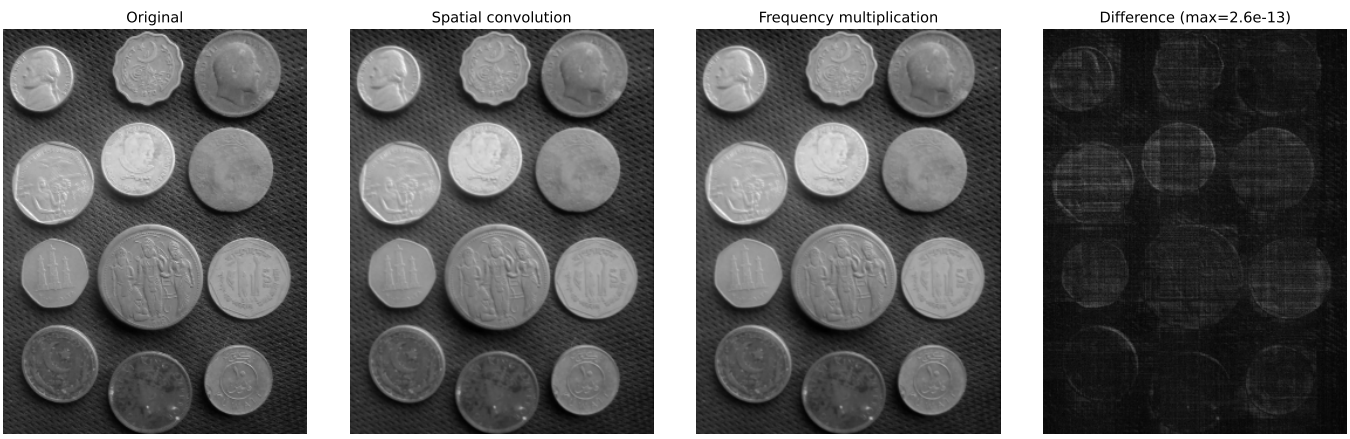


Figure 5.9: Verification of the Convolution Theorem: the pixel-by-pixel difference between spatial convolution (cv2.filter2D) and frequency multiplication (FFT) is numerically zero — confirming the theoretical equivalence.

On Numerical Difference

The residual difference on the order of 10^{-13} does not violate the Convolution Theorem, but rather reflects **computational limitations** inherent to floating-point arithmetic (double precision, $\sim 10^{-16}$) and to the **order of operations** between the two methods:

- **Spatial convolution:** weighted sum of neighbors with successive roundings.
- **Frequency convolution:** involves three FFT transforms and one complex multiplication, subject to truncation and quantization errors.

Therefore, the theoretical equality is exact, but the numerical implementation produces a practically negligible difference (relative error $< 10^{-12}$), confirming the theorem within machine precision.

5.5 Frequency Domain Filters

A frequency domain filter can be interpreted as a **transfer function applied to the image spectrum**. In this representation, each frequency coefficient is multiplied by a value between 0 and 1, which determines its attenuation or preservation. The shape of this function defines the visual effect of the filter.

Abrupt cutoff and *ringing*. Ideal filters with an instantaneous transition at a cutoff frequency D_0 produce discontinuities in the frequency domain. This discontinuity is reflected in the spatial domain as oscillations near edges, known as *ringing*. This effect is associated with convolution with functions of infinite support in space, such as the *sinc* function, as illustrated in Figure 5.10.

Filters with smooth transition. Alternatives such as the Gaussian and Butterworth filters smooth the transition between pass and reject regions, reducing *ringing*. In contrast, this smoothing implies a less defined separation boundary between preserved and attenuated frequencies.

```
# Simulating the Ideal Frequency Filter (Cylinder) and its Spatial Representation (Sinc)
N_grid = 2**7
u = np.arange(-N_grid//2, N_grid//2)
U, V = np.meshgrid(u, u)
D = np.sqrt(U**2 + V**2)

# Frequency: Ideal Cylinder (1 at the center, 0 outside radius 20)
H_freq = np.zeros((N_grid, N_grid))
H_freq[D <= 20] = 1

# Space: The inverse results in the infamous 2D Sinc
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()
```

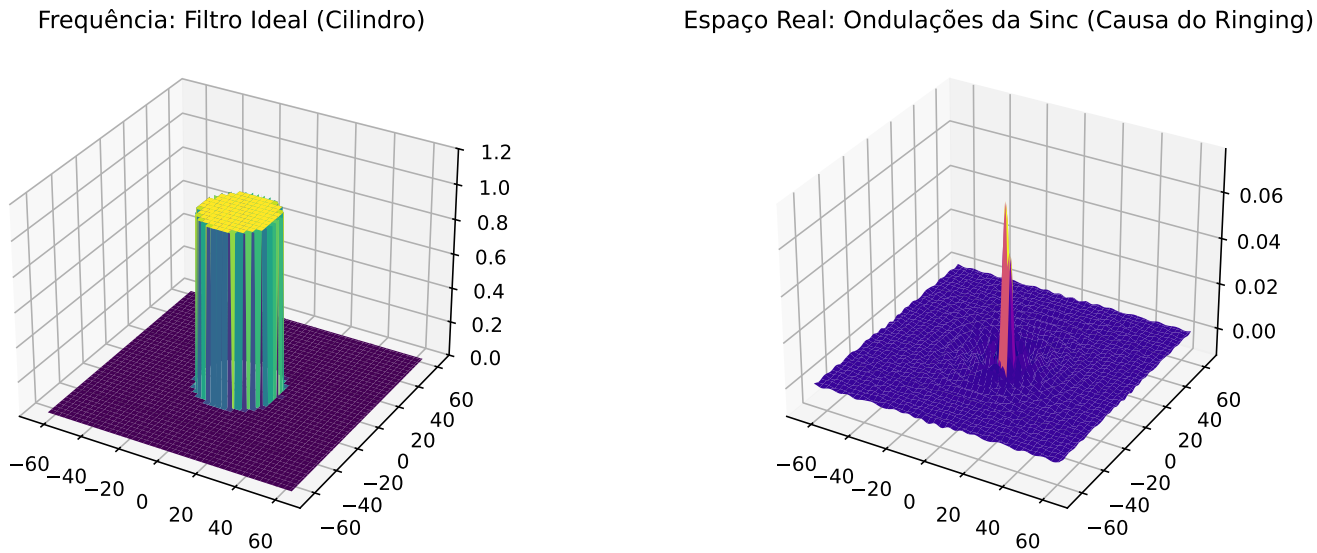


Figure 5.10: The Dangerous Duality: The abrupt cutoff in Frequency (Cylinder) necessarily transforms into a spatial Sinc. Its ripples cause the ghostly *ringing* at the edges of the image.

5.5.1 Low-Pass Filters

Low-pass filters attenuate high-frequency components, resulting in image smoothing and noise reduction. After spectrum centering (FFT Shift), the distance from each point to the center is given by:

$$D(u,v) = \sqrt{\left(u - \tfrac{M}{2}\right)^2 + \left(v - \tfrac{N}{2}\right)^2} \tag{5.5}$$

Ideal Filter (LPFI):

$$H_{\text{ideal}}(u,v) = \begin{cases} 1, & D(u,v) \leq D_0 \\ 0, & D(u,v) > D_0 \end{cases} \tag{5.6}$$

The abrupt cutoff at D_0 introduces discontinuities in the frequency domain, resulting in oscillations in the spatial domain known as *ringing*. This effect is associated with convolution with functions of infinite support.

Gaussian Filter (LPFG):

$$H_{\text{gauss}}(u,v) = e^{-D^2(u,v)/(2\sigma^2)} \tag{5.7}$$

The smoothness of the Gaussian function in the frequency domain avoids discontinuities, which eliminates *ringing* and produces a gradual transition between preserved and attenuated frequencies.

Butterworth Filter (LPFB) of order n:

$$H_{\text{BW}}(u,v) = \frac{1}{1 + [D(u,v)/D_0]^{2n}} \tag{5.8}$$

The parameter n controls the smoothness of the transition between frequency pass and rejection. Small values produce smooth transitions, while large values approximate the behavior of the ideal filter, with a higher risk of *ringing*. A comparative example is presented in Figure 5.11.

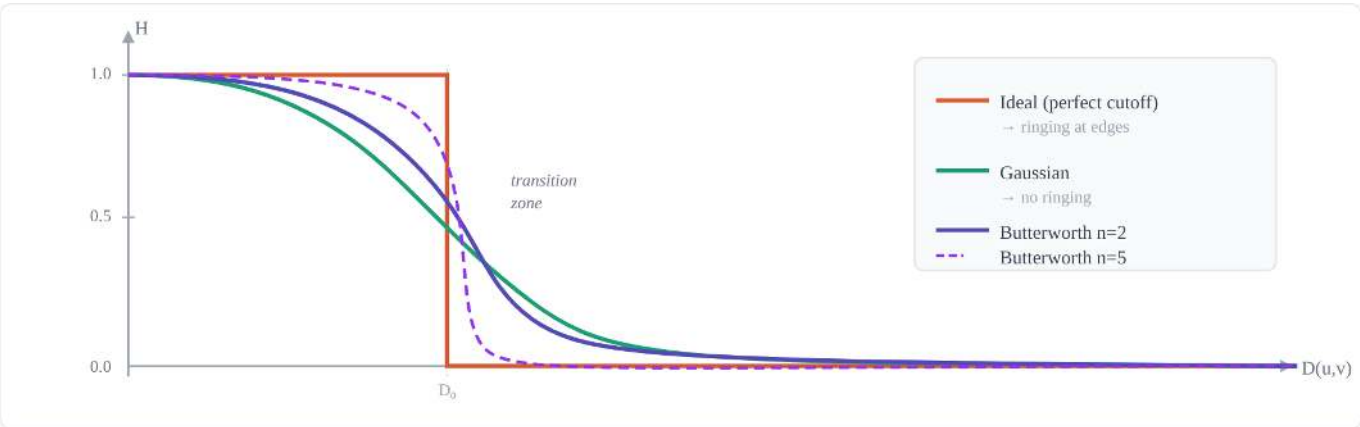


Figure 5.11: Low-pass filters.

5.5.2 High-Pass and Band-Pass Filters

High-pass filters can be obtained from a complementary low-pass filter, defined as:

$$H_{\text{HP}}(u,v) = 1 - H_{\text{LP}}(u,v)$$

This type of filter preserves high-frequency components, enhancing edges and details, while attenuating regions of smooth variation.

Band-pass filters preserve only an intermediate range of frequencies, bounded by two radii D_L and D_H:

$$H_{\text{BP}}(u,v) = H_{\text{LP}}^{(D_H)}(u,v) \cdot \left[1 - H_{\text{LP}}^{(D_L)}(u,v)\right]$$

This type of filtering is useful when one wishes to simultaneously remove low- and high-frequency components, preserving only structures of intermediate scale.

An important application is the removal of **periodic noise**, in which regular patterns appear as localized peaks in the magnitude spectrum. These peaks can be attenuated by means of *notch* (band-reject) filters, positioned specifically at the undesired frequencies.

Examples of filters in the frequency domain are presented in the simulator of Figure 5.12, Figure 5.13, and Figure 5.14.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap05/sim-05-filtros.html

Figure 5.12: Interactive simulator of filters in the frequency domain.

```
import io

def distancia_centro(M, N):
    """Distance matrix from the center of the spectrum."""
    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):
    """Applies filter H (centered) to image 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      # cutoff frequency
n_bw = 2       # Butterworth order

#   Transfer functions
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))

#   Filtered images
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)

#   H(u,v) profiles
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)

#   Visualized H filters
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", "Ideal LPF", "Gaussian LPF", "Butterworth LPF (n=2)",
        "Ideal H",  "Gaussian H","Butterworth H", "H(u,v) Profiles"
    ],
    cols=4, figsize=(16, 9)
)
```

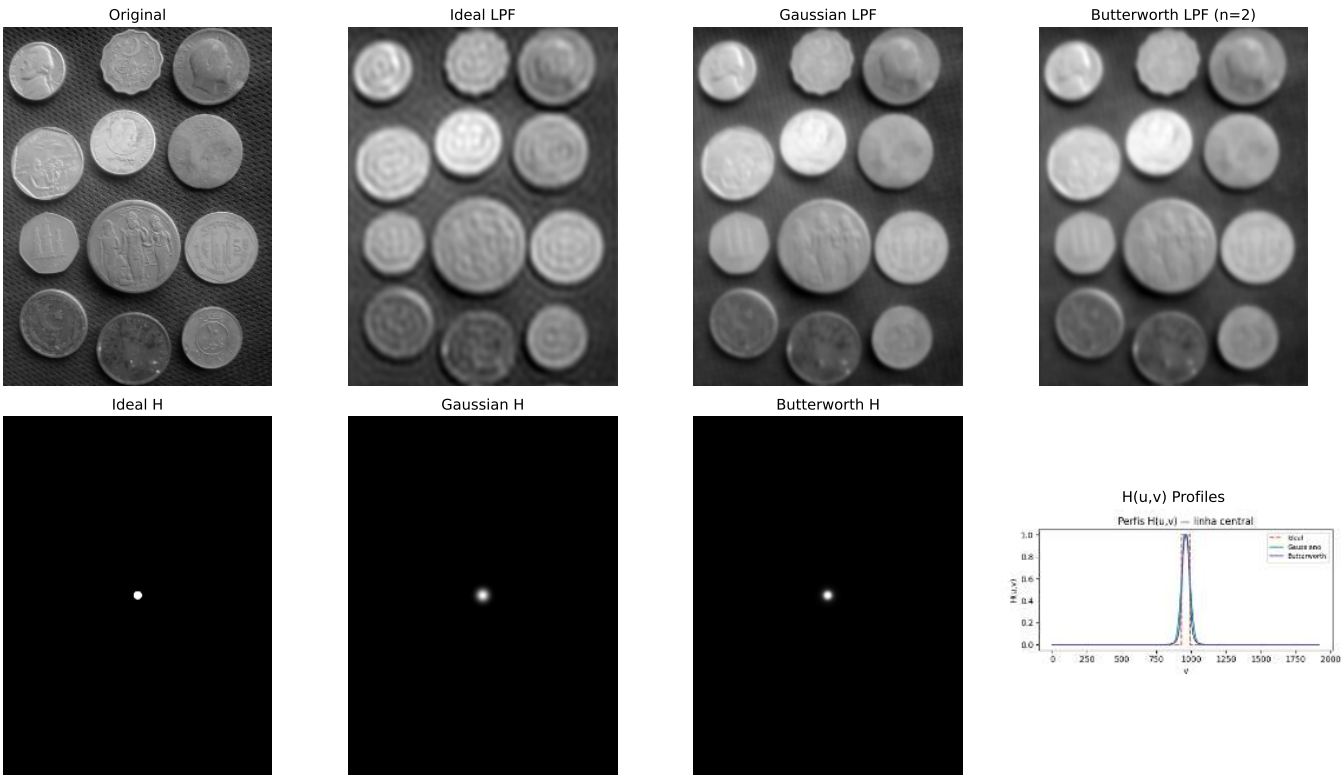


Figure 5.13: Comparison between low-pass filters: Ideal (D =30), Gaussian (D =30) and Butterworth (D =30, n=2). H(u,v) profiles along a central line and corresponding filtered images.

```
# High-pass filter: complement of the Gaussian low-pass
# Reuses aplicar_filtro_freq() defined in the previous cell
H_alta   = 1 - H_gauss
img_alta = aplicar_filtro_freq(img_gray, H_alta)

mm.show(
    [img_gray, img_alta],
    titles=["Original", "Gaussian high-pass ($D_0=30$)"],
    cols=2
)
```

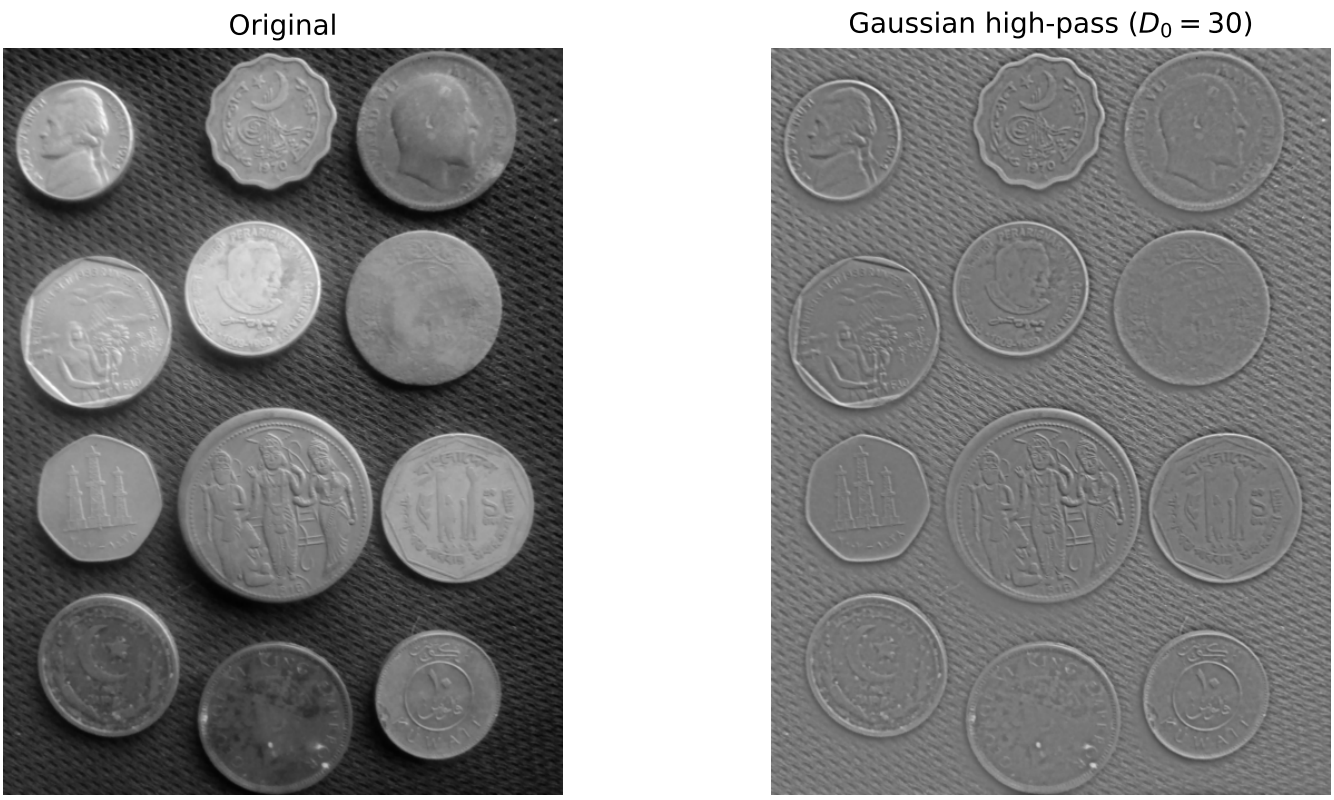


Figure 5.14: Gaussian high-pass filter. (a) Original; (b) High-pass filter (D =30) - the edges of the coins and textured background are enhanced.

5.5.3 Periodic Noise Removal

Periodic noise — associated with electrical interference, regular sensor patterns, or scanning artifacts — appears in the Fourier spectrum as **symmetrical point peaks around the center**.

The **notch filter** selectively attenuates these frequencies while preserving the remaining image components. An application example is presented in Figure 5.15.

```
#   Image with synthetic periodic noise
h_img, w_img = img_gray.shape
x = np.arange(w_img)
y = np.arange(h_img)
X, Y = np.meshgrid(x, y)

# Use integer and consistent frequencies (important for perfect restoration)
u0, v0 = 20, 20  # exact noise frequencies

ruido = 40 * np.sin(2 * np.pi * (u0 * X / w_img + v0 * Y / h_img))
img_ruidosa = np.clip(img_gray.astype(np.float64) + ruido, 0, 255).astype(np.uint8)

#   Spectrum of the noisy image
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)

#   Notch mask
mascara = np.ones((h_img, w_img), dtype=np.float64)
r_notch = 8  # notch radius (fine-tune if necessary)

def suprimir_pico(mask, cy, cx, r):
```

```
    """Zeros a disk of radius r centered at (cy, cx)"""
    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

# Central coordinates
cy, cx = h_img // 2, w_img // 2

# Suppress the 4 symmetric peaks (important!)
for dy, dx in [(v0, u0), (-v0, -u0), (v0, -u0), (-v0, u0)]:
    mascara = suprimir_pico(mascara, cy + dy, cx + dx, r_notch)

mascara_vis = (mascara * 255).astype(np.uint8)

#   Filtering and reconstruction
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)

# Evaluation
psnr = cv2.PSNR(img_gray, img_rest_vis)
ssim = cv2.SSIM(img_gray, img_rest_vis) if hasattr(cv2, 'SSIM') else "N/A"
#ssim = ssim_sk(img_gray, img_rest_vis, data_range=255)

print(f"PSNR (original vs restored): {psnr:.2f} dB")

#   Visualization
mm.show(
    [img_ruidosa, mag_vis, mascara_vis, img_rest_vis],
    titles=[
        "With periodic noise",
        "Spectrum (log)",
        "Notch mask",
        f"Restored (PSNR={psnr:.1f} dB)"
    ],
    cols=4,
    figsize=(16, 4)
)
```

```
PSNR (original vs restored): 32.93 dB
```

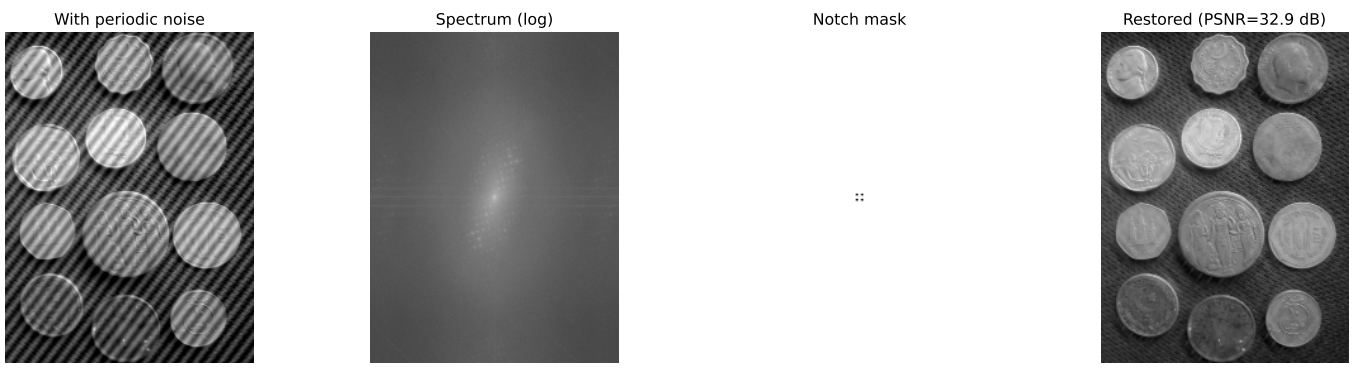


Figure 5.15: Periodic noise removal via *notch* filter in the frequency domain: (a) image with sinusoidal noise, (b) spectrum showing noise peaks, (c) *notch* mask centered on the peaks, (d) restored image.

📌 Synthesis — Spectral Filters

Filter	Visual effect	Artifact	Use
Ideal low-pass	Intense smoothing	*Ringing*	Illustrative
Gaussian low-pass	Gentle smoothing	No *ringing*	General smoothing
Butterworth low-pass	Controlled smoothing	*Ringing* (high orders)	Trade-off between smoothing and selectivity
High-pass	Edge enhancement	Noise amplification	Contour detection
Notch	Selective frequency removal	Possible local distortions	Periodic noise removal

The design of filters in the frequency domain consists of defining spectral masks. However, effects in the spatial domain, such as *ringing* and blurring, emerge directly from these choices in the spectrum.

5.6 *Wavelets* and Multiresolution

The Fourier Transform decomposes the signal into **global** frequencies: each coefficient $F(u, v)$ receives contributions from the entire image, with no explicit information about the spatial localization of those frequencies. Thus, localized structures, such as edges, are represented in a distributed manner across the spectrum.

***Wavelets* (ondaletas)** overcome this limitation by using basis functions **localized in space**, which can be translated and scaled. These functions have **compact support**, that is, they are nonzero only in a finite region of the domain, allowing a simultaneous representation in terms of **frequency and spatial localization**.

5.6.1 The Limit of the Fourier Transform: Spatial Localization

The Fourier Transform accurately describes **which frequencies are present** in a signal, but it does not explicitly represent **where those frequencies occur in space**.

In the experiment presented in Figure 5.16, two images with structures located at different positions produce nearly identical magnitude spectra. This occurs because the Fourier representation is global: each coefficient receives contributions from the entire image.

As a result, the magnitude spectrum does not explicitly represent the location of edges or other structures, only the distribution of the frequencies present. This limitation motivated the development of multiresolution representations, such as the Discrete *Wavelet* Transform (DWT), capable of simultaneously describing the frequency and spatial localization of image structures.

```
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=["Signal A", "Spectrum A", "Signal B (Shifted)", "Spectrum B"],
        cols=4, figsize=(14, 4)
        )
```

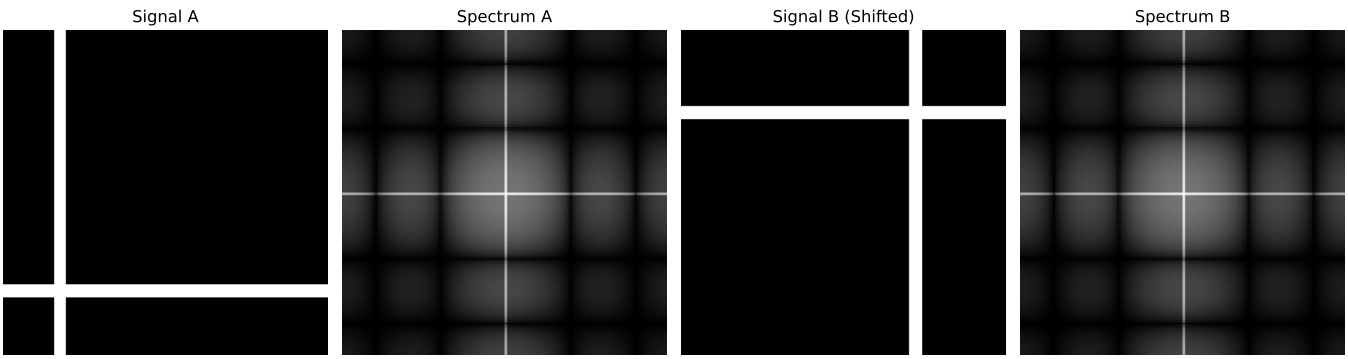


Figure 5.16: Fourier global is blind to position. The spectra do not tell where the edges are.

5.6.2 2D Discrete Wavelet Transform

The Discrete Wavelet Transform (DWT) applies, separately in the horizontal and vertical directions, two complementary filters: a **low-pass** h (approximation) and a **high-pass** g (details), followed by downsampling by a factor of 2 in each dimension. This process produces four subbands, whose names indicate the combination of filters applied in each direction (L = *Low-pass*; H = *High-pass*). The characteristics of each subband are summarized in Table 5.3.

$$\mathrm{DWT}(f) = \{ \underbrace{\mathrm{LL}}_{\text{aprox.}}, \underbrace{\mathrm{LH}}_{\text{horizontal details}}, \underbrace{\mathrm{HL}}_{\text{vertical details}}, \underbrace{\mathrm{HH}}_{\text{diagonal details}} \}.$$

Table 5.3: Subbands produced by the 2D Discrete Wavelet Transform (DWT), indicating the filters applied in each direction and the predominant content of each component.

Subband	Filters applied	Visual content
LL	low × low	Image approximation (smoothed and reduced version)
LH	low × high	Horizontal edges and vertical variations
HL	high × low	Vertical edges and horizontal variations
HH	high × high	Diagonal details and textures

The decomposition can be applied recursively to the LL subband, generating a multiresolution representation. After J levels, a structure with $3J + 1$ subbands is obtained, where each new level reduces the resolution of the approximation component.

> **Connection with CNNs**
>
> The multiresolution decomposition of wavelets has a conceptual relationship with the hierarchical representations used in convolutional neural networks (CNNs). In both cases, successive stages of filtering and resolution reduction produce increasingly abstract descriptions of the image. However, **wavelets use mathematically defined and reconstructible filters**, whereas **CNNs learn their filters during training**.

5.6.3 Wavelet Families

Different wavelet families present distinct trade-offs between **spatial support**, smoothness, and compression capability. **Support** corresponds to the extent of the wavelet function in the spatial domain: the smaller the

support, the more localized the function; the larger it is, the smoother its representation tends to be, albeit at a higher computational cost. Table 5.4 compares some of the most commonly used families.

Table 5.4: Comparison among wavelet families, highlighting support length, number of vanishing moments, symmetry, and typical applications.

Wavelet	Support length	Vanishing moments	Symmetry	Typical use
Haar	2	1	Asymmetric	Introduction and basic analysis
Daubechies db4	8	4	Asymmetric	Compression and general analysis
Symlet sym4	8	4	Nearly symmetric	Signal reconstruction
Biorthogonal 5/3	5/3	2/2	Symmetric	Lossless JPEG 2000
Biorthogonal 9/7	9/7	4/4	Symmetric	Lossy JPEG 2000

Vanishing moments measure the wavelet's ability to represent smooth regions of the image with few nonzero coefficients. A wavelet with p vanishing moments exactly annihilates polynomials of degree up to $p-1$. Consequently, the greater the number of vanishing moments, the greater the compression efficiency tends to be in homogeneous regions, although this generally implies functions with longer support.

Figure 5.17 presents the basis functions (wavelets) $\psi(t)$ in the spatial domain. These functions have **compact support**, that is, they are nonzero only in a finite region of the domain, unlike the sinusoids of the Fourier Transform, which extend over the entire domain.

```
#   Import pywt
try:
    import pywt
except ImportError:
    import subprocess
    subprocess.run(["pip", "install", "PyWavelets", "-q"])
    import pywt

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()
```

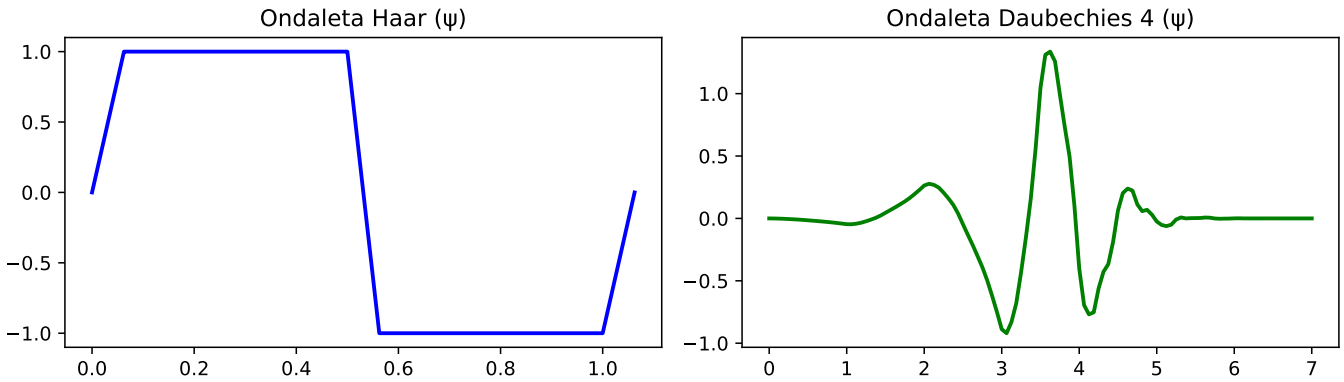


Figure 5.17: Wavelet functions ( ). Note how they quickly decay to zero (compact support), unlike the infinite sinusoids of Fourier.

The diagram in Figure 5.18 illustrates the multiresolution analysis performed by the DWT, in which the approximation subband (LL) is successively decomposed, forming a hierarchical representation with two levels.

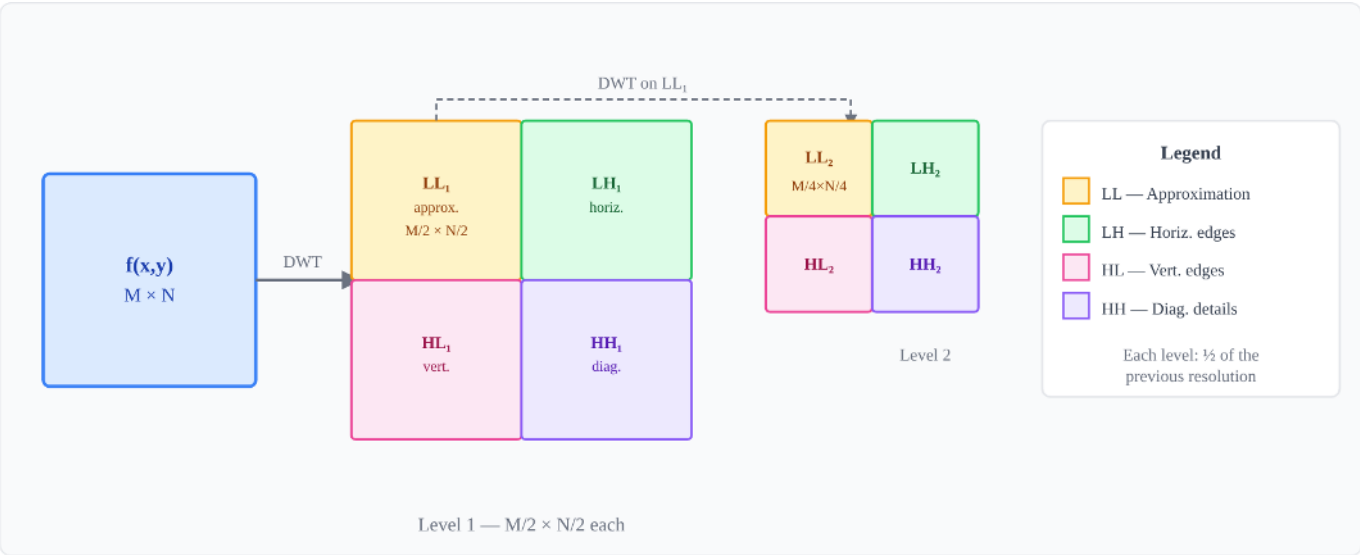


Figure 5.18: Diagram of the 2D wavelet decomposition at two levels.

The simulator in Figure 5.19 allows for interactive exploration of the 2D Discrete *Wavelet* Transform (DWT) using the Haar *wavelet*. The subband decomposition highlights the separation between the approximation component and the detail components of the image.

The different input patterns allow observing the directional behavior of the filters. In images with horizontal and vertical edges, the LH and HL subbands highlight, respectively, the vertical and horizontal intensity variations. In regions of smooth variation, most of the energy is concentrated in the LL approximation subband, while the detail subbands exhibit coefficients close to zero.

Multiresolution analysis can also be observed by increasing the number of decomposition levels. In this case, only the LL_1 subband is further decomposed, giving rise to the LL_2, LH_2, HL_2, and HH_2 subbands, which form the second level of the hierarchical representation.

In patterns composed of large homogeneous regions, such as a smooth gradient or a checkerboard made of large blocks, the energy remains predominantly concentrated in the LL subband. In the gradient, this occurs because the differences between neighboring pixels are small. In the checkerboard, in turn, the pixels have practically the same intensity within each block, so that only the boundaries between blocks produce nonzero coefficients in the detail subbands. Since these boundaries occupy only a small fraction of the image, their

contribution to the total energy remains reduced.

To enable the visual analysis of these subtle variations, the simulator incorporates a contrast gain control for the details (ranging from 1 to 8). This parameter functions as a linear amplification factor applied exclusively to the coefficients of the detail subbands (LH, HL, and HH) before their on-screen rendering. In scenarios of smooth transition (such as the gradient) or local uniformity (such as the interior of the checkerboard blocks), the numerical differences calculated by the Haar high-pass filter result in coefficients very close to zero, which would render the corresponding quadrants dark and imperceptible to the naked eye. By multiplying these values by the gain, the simulator visually recovers the hidden high-frequency structures and enhances the orientation of the remaining edges.

The energy-per-subband plot quantifies this distribution between the approximation component and the detail components, demonstrating that the visual gain does not alter the original energy metric. In natural images, most of the energy is concentrated in the LL subband, while the LH, HL, and HH subbands mainly represent edges, textures, and other local intensity variations.

5.6.4 Multiresolution Analysis with the 2D DWT

The 2D Discrete Wavelet Transform (DWT) decomposes an image into approximation and detail components, organized hierarchically across different scales and orientations. As detail subbands in natural images often exhibit low-contrast coefficients, the practical examples below use a synthetic geometric pattern generated in Python. This approach replicates the behavior of the simulator in Figure 5.19, making the effects of spatial filtering and multiresolution decomposition visually explicit.

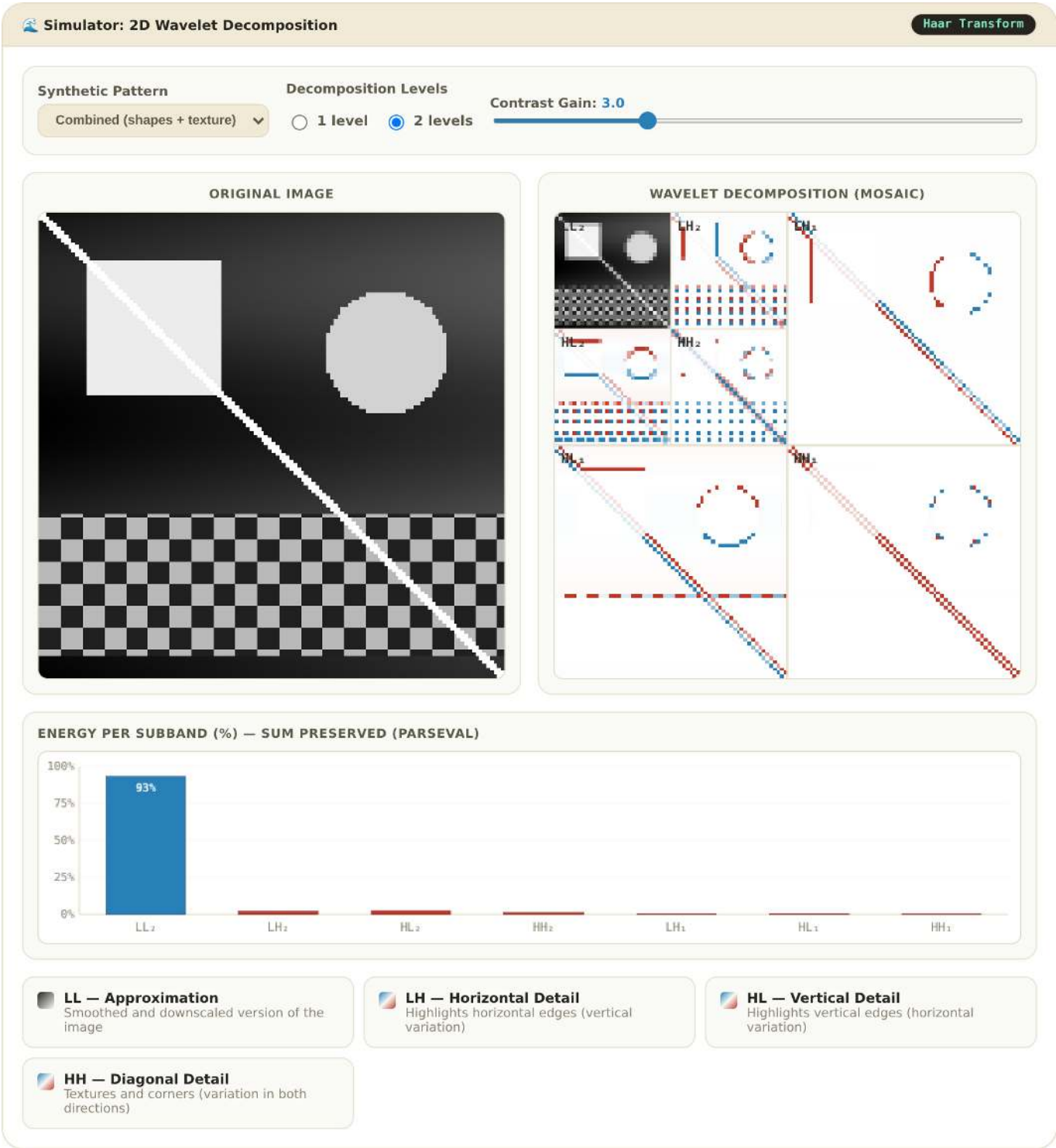
5.6.4.1 Multi-Level Mosaic Decomposition

Figure 5.20 illustrates the hierarchical structure of the two-level DWT using the Haar *wavelet*. The process is based on the combined application of low-pass and high-pass filters in the horizontal and vertical directions, followed by subsampling by a factor of 2.

In the first level, the original image yields the approximation subband (LL_1) and the horizontal (LH_1), vertical (HL_1), and diagonal (HH_1) detail components. In multiresolution analysis, the LL_1 subband is again filtered and subsampled, generating the second decomposition level (LL_2, LH_2, HL_2, and HH_2).

To enable the visual interpretation of the detail components, the code extracts the absolute value of their coefficients and applies linear (*min-max*) normalization to occupy the entire dynamic range of gray levels [0, 255]. This operation transforms homogeneous regions (zero coefficients) into black and highlights in white the edges and textures extracted at each scale and orientation.

```
try:
    import pywt
    HAS_PYWT = True
except ImportError:
    import subprocess
    subprocess.run(["pip", "install", "PyWavelets", "-q"])
    import pywt
    HAS_PYWT = True

import numpy as np
import cv2

#   Generation of the Synthetic Image (Same 'combined' pattern as the simulator)
def gerar_imagem_sintetica(N=256):
    img = np.zeros((N, N), dtype=np.float64)
    for y in range(N):
        for x in range(N):
            v = 55 + 35 * (x / N) + 15 * np.sin(y / 24)
            # Square
            if 24 < x < 100 and 24 < y < 100:
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap05/sim-05-wavelet.html

Figure 5.19: Simulation of 2D *wavelet* decomposition.

```
                v = 225
            # Circle
            cx, cy, r = 190, 76, 34
            if (x - cx)**2 + (y - cy)**2 < r**2:
                v = 205
            # Periodic texture (bottom)
            if y > 164 and y < 244:
                p = 12
                v = 185 if ((x // p + y // p) % 2 == 0) else 65
            # Diagonal line
            if abs(x - y) < 4:
                v = 240
            img[y, x] = np.clip(v, 0, 255)
    return img.astype(np.uint8)

# Replaces the dark coins image with the clear synthetic pattern
img_gray = gerar_imagem_sintetica(256)

#   2-level wavelet decomposition
wavelet   = "haar"
img_float = img_gray.astype(np.float64)

# Level 1
coefs1 = pywt.dwt2(img_float, wavelet)
LL1, (LH1, HL1, HH1) = coefs1

# Level 2 (applied on LL1)
coefs2 = pywt.dwt2(LL1, wavelet)
LL2, (LH2, HL2, HH2) = coefs2

def sb_vis(sb):
    """Normalizes subband for visualization [0,255]."""
    return cv2.normalize(np.abs(sb), None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

print(f"Original shape     : {img_gray.shape}")
print(f"LL1 (level 1)      : {LL1.shape}  |  LH1/HL1/HH1: {LH1.shape}")
print(f"LL2 (level 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))
```

```
Original shape     : (256, 256)
LL1 (level 1)      : (128, 128)  |  LH1/HL1/HH1: (128, 128)
LL2 (level 2)      : (64, 64)    |  LH2/HL2/HH2: (64, 64)
```

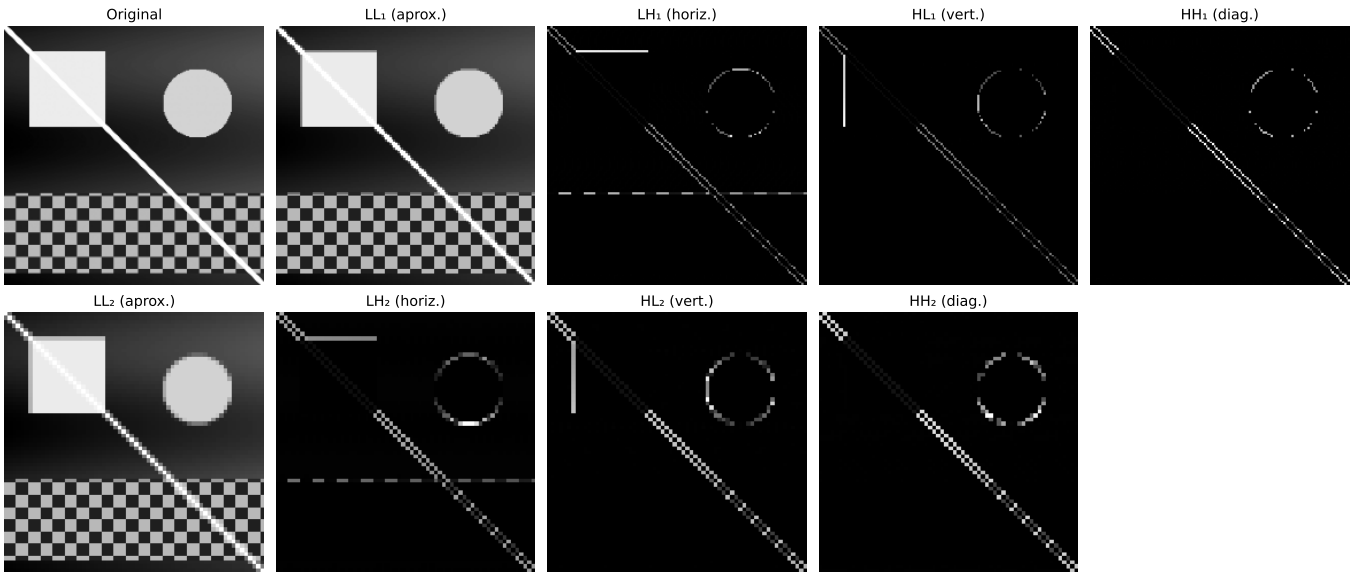


Figure 5.20: 2D *wavelet* decomposition of 2 levels with Haar *wavelet*: LL, LH, HL, HH subbands at each level. The detail subbands reveal oriented structures at different scales using a synthetic pattern.

5.6.4.2 The Trade-off Between Localization and Smoothness

The choice of the basis function (wavelet) directly influences how image features are distributed and encoded by the DWT coefficients. Figure 5.21 compares the practical results obtained by applying four distinct families to the synthetic geometric pattern: `haar`, `db4`, `sym4`, and `bior2.2`.

Due to its short support and step-function shape, the Haar wavelet produces coefficients that are highly localized at spatial discontinuities, generating thin, sharp edges in the detail subbands. In contrast, families such as Daubechies (`db4`) and Symlets (`sym4`), which exhibit larger support (longer filters) and a greater number of vanishing moments, yield smoother, more distributed responses around transitions, which can introduce slight oscillations or blurring at abrupt boundaries.

This behavior highlights the classic trade-off in multiresolution analysis: smaller supports favor precise spatial localization of edges, while larger supports and a greater number of vanishing moments tend to produce sparser, smoother representations. This smoothness and ability to attenuate high frequencies ensure greater efficiency in energy compaction—fundamental characteristics for data compression and denoising applications.

```
import numpy as np
import cv2
import pywt

# Ensures that img_gray and img_float use the same clear synthetic pattern
if 'gerar_imagem_sintetica' in globals():
    img_gray = gerar_imagem_sintetica(256)
else:
    # Fallback in case the previous block was not executed in the same session
    def gerar_imagem_sintetica(N=256):
        img = np.zeros((N, N), dtype=np.float64)
        for y in range(N):
            for x in range(N):
                v = 55 + 35 * (x / N) + 15 * np.sin(y / 24)
                if 24 < x < 100 and 24 < y < 100: v = 225
                cx, cy, r = 190, 76, 34
                if (x - cx)**2 + (y - cy)**2 < r**2: v = 205
                if y > 164 and y < 244:
                    p = 12
```

```
                    v = 185 if ((x // p + y // p) % 2 == 0) else 65
                if abs(x - y) < 4: v = 240
                img[y, x] = np.clip(v, 0, 255)
        return img.astype(np.uint8)
    img_gray = gerar_imagem_sintetica(256)

img_float = img_gray.astype(np.float64)

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))
```

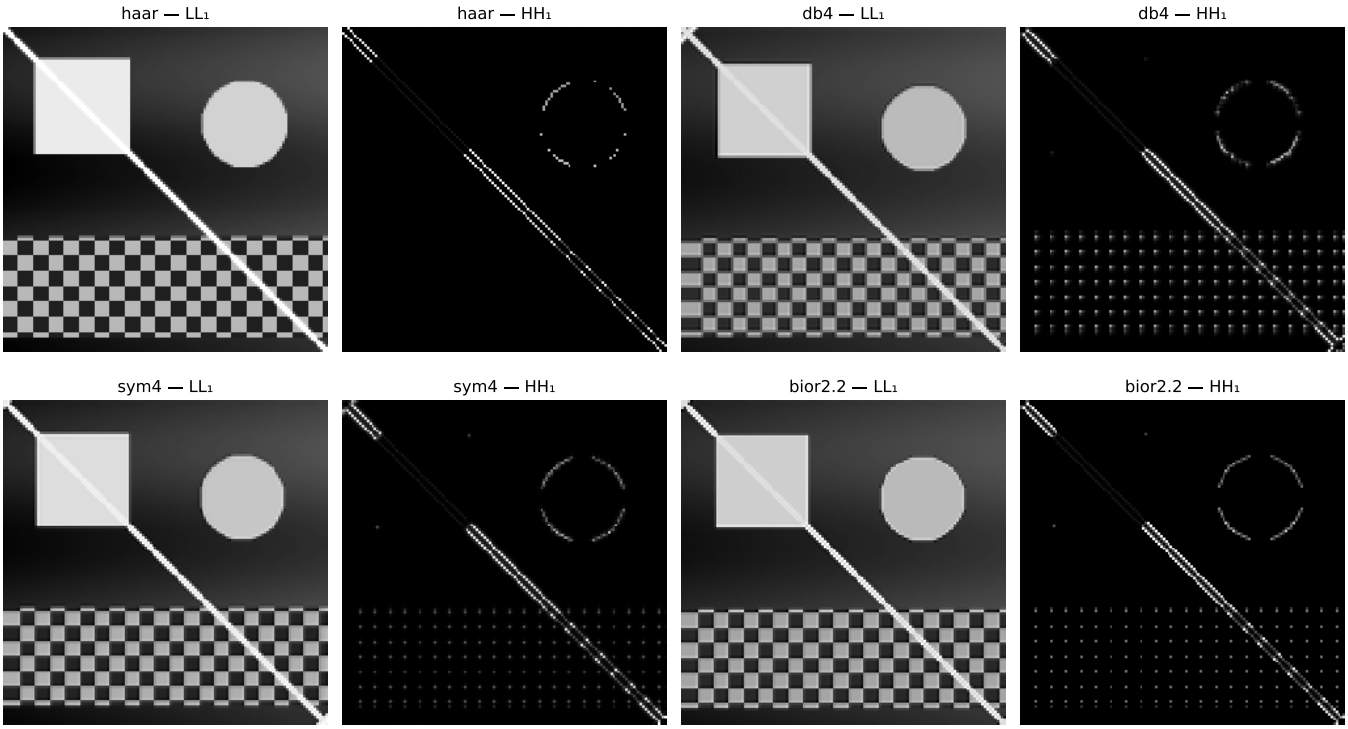


Figure 5.21: Comparison between *wavelet* families: Haar, db4, sym4, and bior2.2. LL subband (approximation) and HH (diagonal) for each choice, illustrating the trade-off between compactness and smoothness based on the synthetic pattern.

5.6.4.3 Coefficient Thresholding and Compression

One of the main applications of the Discrete *Wavelet* Transform (DWT) is data compression, driven by the **sparse** representation capability of the coefficients. Figure 5.22 illustrates the effect of hard thresholding, a technique in which detail coefficients with magnitude below a threshold T are entirely zeroed out before the synthesis process carried out by the Inverse Discrete *Wavelet* Transform (IDWT).

As the threshold T is increased, a growing number of high-frequency coefficients are zeroed. Since they concentrate less energy, removing these components considerably reduces the amount of information required to represent the image, while keeping the global approximation component (the deepest LL subband) intact to preserve the macro structure. Visually, this discarding of coefficients manifests itself through the progressive disappearance of fine textures and the smoothing of abrupt intensity transitions.

The fidelity of the reconstructed image relative to the original is quantified by the **Peak Signal-to-Noise Ratio (PSNR)** metric, expressed in decibels (dB). Higher PSNR values indicate less distortion and greater mathematical proximity to the original signal. The practical experiment highlights the gradual decay of PSNR as the thresholding aggressiveness increases, allowing the optimal threshold for the balance between compression and visual degradation to be numerically evaluated.

```
import numpy as np
import cv2
import pywt

# Ensures img_gray uses the same clear synthetic pattern
if 'gerar_imagem_sintetica' in globals():
    img_gray = gerar_imagem_sintetica(256)
else:
    # Fallback in case the previous block was not executed in the same session
    def gerar_imagem_sintetica(N=256):
        img = np.zeros((N, N), dtype=np.float64)
        for y in range(N):
            for x in range(N):
                v = 55 + 35 * (x / N) + 15 * np.sin(y / 24)
                if 24 < x < 100 and 24 < y < 100: v = 225
                cx, cy, r = 190, 76, 34
                if (x - cx)**2 + (y - cy)**2 < r**2: v = 205
                if y > 164 and y < 244:
                    p = 12
                    v = 185 if ((x // p + y // p) % 2 == 0) else 65
                if abs(x - y) < 4: v = 240
                img[y, x] = np.clip(v, 0, 255)
        return img.astype(np.uint8)
    img_gray = gerar_imagem_sintetica(256)

def dwt_threshold_reconstruct(img, wavelet='db4', nivel=2, threshold=0.0):
    """Decomposes, applies threshold and reconstructs via IDWT."""
    coefs = pywt.wavedec2(img.astype(np.float64), wavelet, level=nivel)
    # Copies and applies hard thresholding to all details
    coefs_t = [coefs[0]]   # Final LL is not thresholded
    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)
    # Crop to original dimension
    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))
```

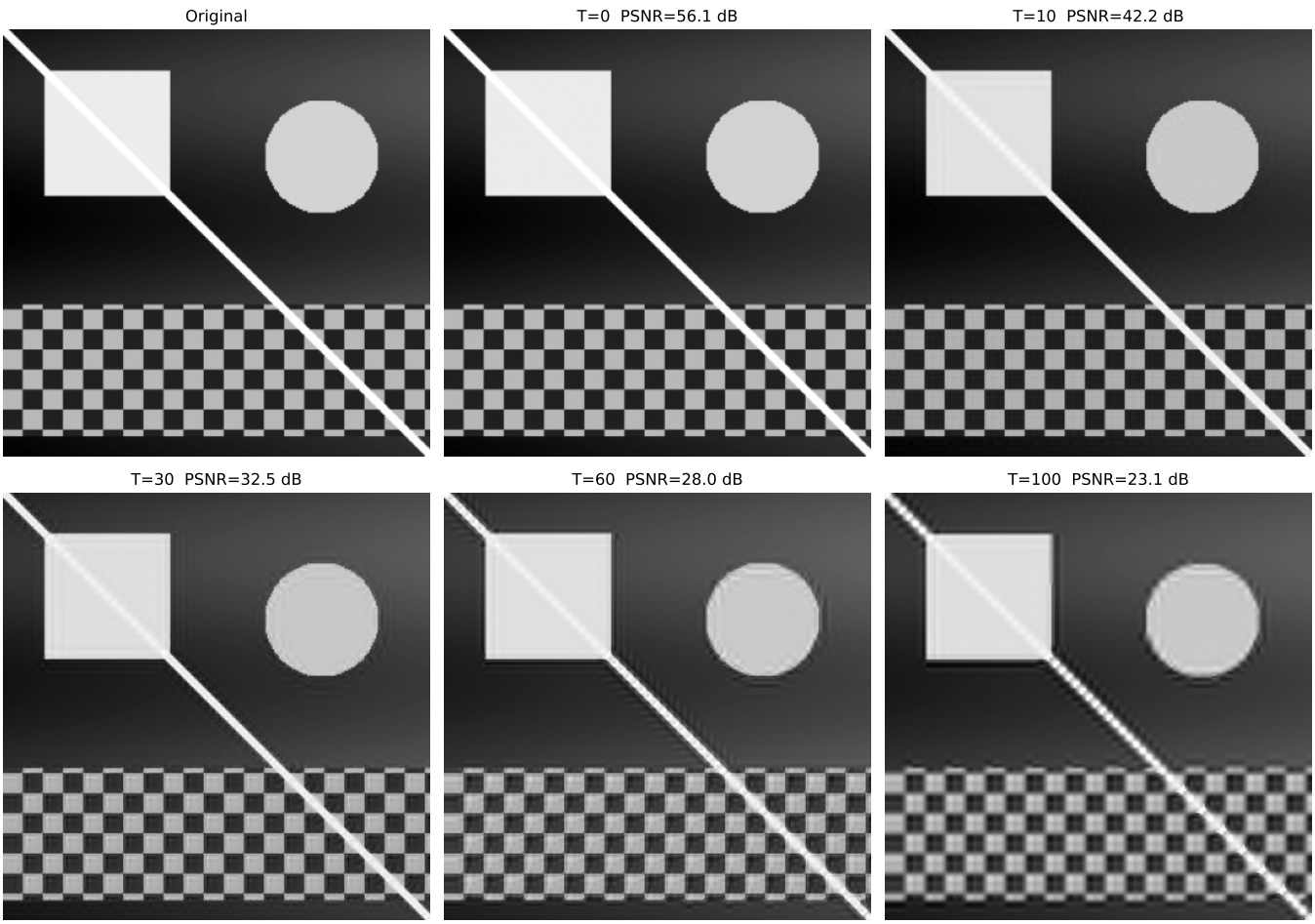


Figure 5.22: Wavelet reconstruction with coefficient thresholding (*hard thresholding*): as the threshold increases, more details are zeroed out, producing progressively smoother images. The PSNR metric quantifies the quality loss on the synthetic pattern.

Synthesis — Fourier vs. *Wavelets*: when to use each approach?

Table 5.5 summarizes the main structural and operational differences between the Discrete Fourier Transform (DFT) and the Discrete Wavelet Transform (DWT).

Table 5.5: Comparison between the Discrete Fourier Transform (DFT) and the Discrete Wavelet Transform (DWT), highlighting their main characteristics and applications.

Criterion	Fourier (DFT)	*Wavelet* (DWT)
Basis functions	Sinusoids with infinite support	Functions with compact support
Spatial localization	Not explicit (global)	Explicit (local)
Spectral filtering	Excellent for fine frequency control	Subband-based (scales)
Image compression	Basis of DCT (traditional JPEG)	Basis of DWT (JPEG 2000)
Multiscale analysis	No	Yes
Periodic noise removal	Highly efficient	Not recommended
Non-stationary signals	Limited	Highly efficient

In practical terms, the DFT stands out as the ideal tool for pure spectral analysis, design of selective filters in the frequency domain, and attenuation of periodic and harmonic noises. On the other hand, the DWT excels in scenarios that require rigorous preservation of the spatial localization of features associated with their frequency content, standing out in data compression, multiresolution analysis, and processing of abrupt

transitions. Thus, both transforms should be understood as perfectly complementary techniques, mapping distinct and specific paths for solving problems in DIP-CV.

> **Analogies with Audio: Limitations and Cautions**
>
> When drawing analogies between image and audio processing, it is important to consider the fundamental differences:
> - In stereo/multichannel audio systems, the phase between channels is crucial for the perception of spatial location (interaural phase and time differences).
> - In monaural systems, phase has limited perceptual influence — the human ear is relatively insensitive to the absolute phase of isolated sinusoidal components.
> - In images, the DFT phase is always fundamental for the spatial location of structures, regardless of whether it is a monochromatic or color image.
>
> The analogy between phase in audio and phase in images should be used with caution, emphasizing that, although both carry information about the spatial/temporal organization of the signal, the perceptual mechanisms are fundamentally different.

5.7 Image Compression

While *wavelets* establish the theoretical foundation of the JPEG 2000 standard, the traditional JPEG standard is based on the **Discrete Cosine Transform (DCT)**. Despite structural differences, both approaches share the same fundamental principle: compacting image energy into a reduced number of coefficients and discarding the least relevant components with minimal visual impact.

The central goal of compression is to reduce the volume of data required for storing or transmitting an image. This process is made feasible by identifying and eliminating structural and perceptual **redundancies**.

5.7.1 Taxonomy of Redundancies

The development of compression algorithms is based on the identification and elimination of three main categories of redundancy, summarized in Table 5.6.

Table 5.6: Categories of redundancy in digital images and their respective exploration mechanisms.

Type	Definition	Exploration Approach
Spatial (interpixel)	High correlation and statistical dependence between neighboring pixels.	DCT, DWT, and predictive coding.
Spectral (interchannel)	Statistical correlation between the color channels of the same image.	Color space transformations (e.g., RGB to YC_bC_r).
Psychovisual	Insensitivity of the human visual system (HVS) to high-frequency and low-contrast variations.	Selective coefficient quantization processes.

Depending on the preservation of the original information after the decoding process, compression methods are divided into two fundamental classes:

- **Lossless:** Guarantees a bit-for-bit identical reconstruction of the original image. It is employed in scenarios where data integrity is strictly critical, such as in medical imaging, diagnostic imaging, and storage of textual documents.
- **Lossy:** Admits the introduction of controlled distortion in the signal in exchange for substantially higher compression rates. It is the standard approach for consumer photographs and video streaming, ecosystems in which the HVS tolerates small high-frequency attenuations without perceiving degradation of visual quality.

5.7.2 Discrete Cosine Transform (2D DCT-II)

The **Discrete Cosine Transform** (DCT) constitutes the central operation of the JPEG standard. Unlike the DFT, which uses a complex basis, the DCT is based on purely real trigonometric functions. For an image block $f(x, y)$ of dimensions $N \times N$, the **2D DCT-II** maps the spatial signal to the spatial frequency domain, generating the coefficient matrix $C(u, v)$ through:

$$C(u, v) = \alpha(u)\,\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y)\, \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right] \tag{5.9}$$

where the orthogonal normalization factors are given by $\alpha(0) = \sqrt{1/N}$ and $\alpha(k) = \sqrt{2/N}$ for $k > 0$.

Each coefficient $C(u, v)$ quantifies the contribution — or "weight" — of a specific spatial frequency within that block. The term $C(0, 0)$ is called the **DC component** and represents the average intensity of the block (zero frequency). The remaining coefficients, known as **AC components** (*Alternating Current*), correspond to progressively higher spatial frequencies.

5.7.3 The Basis Functions of the DCT

From a geometric perspective, Equation 5.9 performs the projection of the pixel block onto a set of orthogonal functions. For the standard JPEG case ($N = 8$), the spatial block is decomposed into a linear combination of **64 two-dimensional basis functions**, denoted by $B_{u,v}(x, y)$ and generated by the product of cosine functions:

$$B_{u,v}(x, y) = \cos\left[\frac{\pi(2x+1)u}{16}\right] \cos\left[\frac{\pi(2y+1)v}{16}\right]$$

Thus, the inverse operation can be interpreted as the exact reconstruction of the original block through the weighted sum of these 64 basis matrices, where each coefficient $C(u, v)$ acts as the analytical weight of its respective harmonic component.

The **spatial frequency** indicated by the indices (u, v) determines the number of oscillation cycles along the horizontal and vertical dimensions of the block. As illustrated in Figure 5.23 — whose code isolates each basis by applying the inverse transformation to unit impulses —, these 64 functions are organized into an 8×8 matrix. The upper-left corner ($u = 0, v = 0$) displays the uniform zero-frequency (DC) pattern, while moving to the right (the u axis) or downward (the v axis) maps progressively larger harmonic variations, representing rapid transitions, edges, and textures in the horizontal, vertical, and diagonal orientations.

ℹ DCT vs. DFT: Advantage of Energy Compaction

Both the DCT and the DFT map a spatial $N \times N$ block into a coefficient matrix of the same dimensions. However, for natural images, the DCT exhibits greater efficiency in **energy compaction** at low frequencies. This occurs because the DCT implicitly assumes an even symmetry of the signal at the block boundaries, which is equivalent to a continuous periodic extension, minimizing the effect of spectral spreading (*ringing*). As a result, most AC coefficients decay rapidly to values close to zero, optimizing the compression *pipeline* without introducing perceptible visual degradation.

```
from scipy.fft import dct, idct  # added line

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):
        coef = np.zeros((8, 8)); coef[i, j] = 1
        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()
```

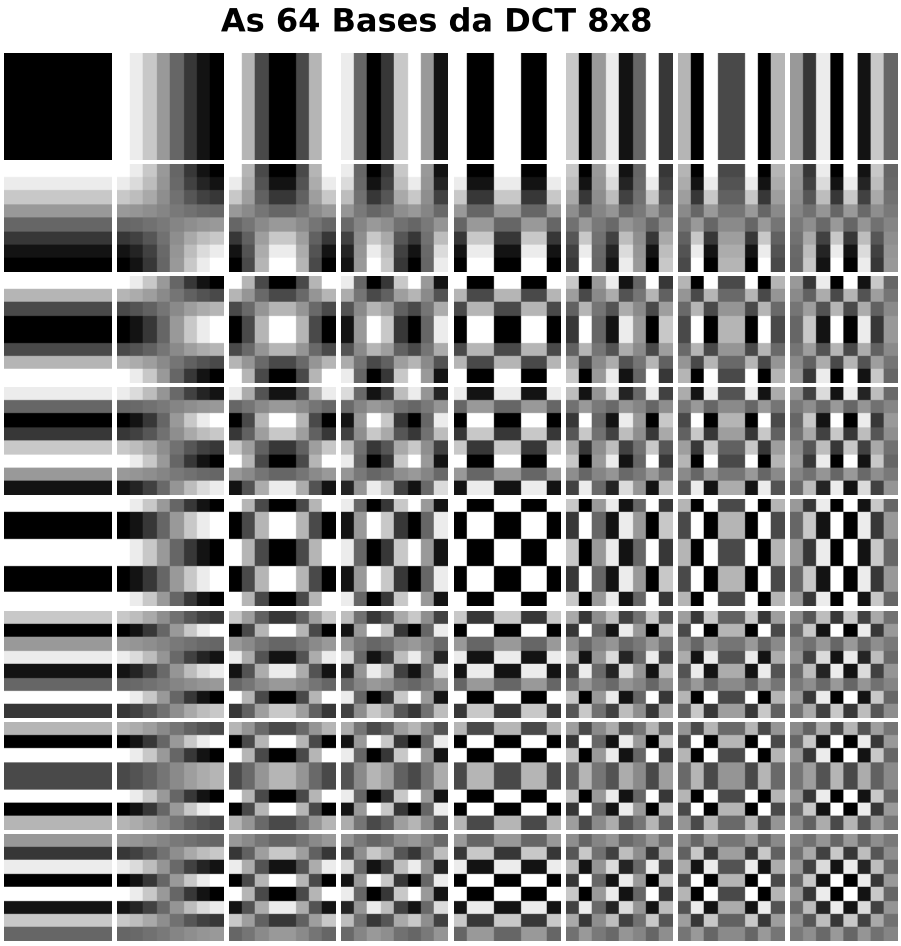


Figure 5.23: The Visual Alphabet of JPEG: The 64 basis functions of the DCT-II. The DC coefficient is at the top left (smooth). Going down and to the right, the spatial oscillation increases drastically.

5.7.4 Energy Concentration and Progressive Reconstruction

Before applying the DCT, the pixels of the intensity block are routinely shifted (by subtracting 128 for 8-bit images) in order to center the signal around zero, eliminating unnecessary DC components. When computing the DCT on the resulting block, the property of **energy compaction** becomes evident: almost all of the variance and information from the original image is concentrated in the DC coefficient ($C(0,0)$) and in the first low-frequency AC harmonics.

Figure 5.24 demonstrates this phenomenon through a progressive reconstruction by abrupt truncation. Instead of using all 64 coefficients, the algorithm preserves only the first k components—selected based on a scan that prioritizes low spatial frequencies—and nullifies the remaining ones.

The inverse synthesis (**IDCT**) performed with only a fraction of the coefficients (such as 15% or 30%) is already capable of recovering the structures and the macro illumination of the original pixel block. As higher-frequency harmonics are progressively reintroduced, fine details and rapid transitions are restored. This behavior validates the principle of perceptual compression: the discarded high frequencies carry little energy, and their absence, under normal conditions, has a secondary visual impact on the observer's perception.

```python
from scipy.fft import dct, idct

def dct2(bloco):
    """Orthogonal 2D DCT-II (separable)."""
    return dct(dct(bloco.T, norm='ortho').T, norm='ortho')

def idct2(coefs):
    """Orthogonal 2D IDCT-II."""
    return idct(idct(coefs.T, norm='ortho').T, norm='ortho')

#   Centralized 8×8 block of the image
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("DCT coefficients of the 8×8 block:")
print(np.round(C).astype(int))
print(f"\nDC energy     : {C[0,0]**2:.1f}")
print(f"Total energy  : {(C**2).sum():.1f}")
print(f"DC fraction   : {C[0,0]**2 / (C**2).sum():.1%} ← energy concentration")

#   Progressive reconstruction
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))
```

```
DCT coefficients of the 8×8 block:
[[ 450    2 -199   -1    0    0  -14    0]
 [  -2  506    0  -58    0   24    0  -24]
 [-199    2   89   -1   82    1    0    0]
 [  -1  -58    0 -158    0   59    0  -24]
 [   0    0   83    0  -89    0  -34    0]
 [   0   24    0   58    0   84    0  -58]
 [ -14    0    0    0  -34    0   89   -1]
 [   0  -24    0  -24    0  -58    0  -76]]

DC energy     : 202725.1
Total energy  : 639868.0
DC fraction   : 31.7% ← energy concentration
```

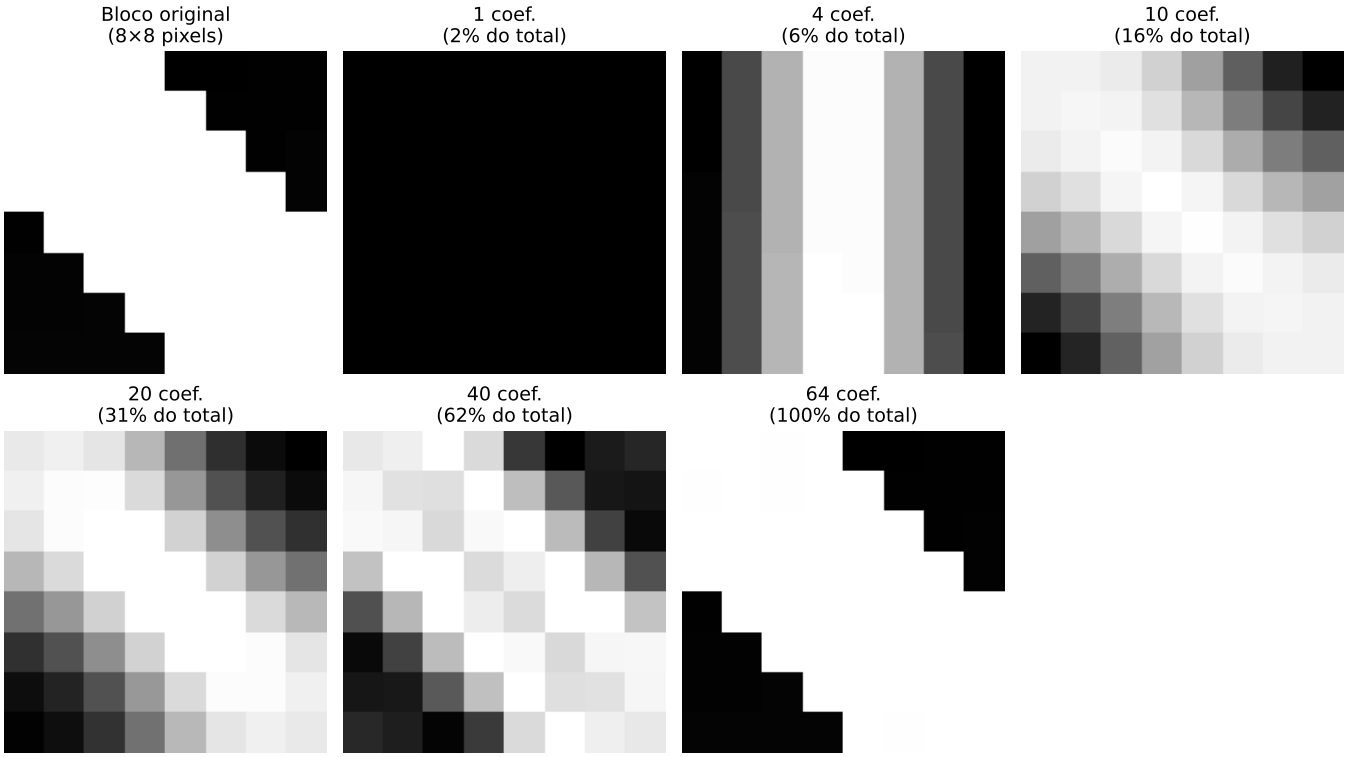


Figure 5.24: 2D DCT in 8×8 block: coefficients and progressive reconstruction.

5.7.5 The JPEG Compression Pipeline

The JPEG standard operates by dividing the image into disjoint blocks of 8×8 pixels, processed through a sequence of spatial, perceptual, and statistical transformations. The complete encoding pipeline is structured into six main stages:

$$\text{RGB} \xrightarrow{(1)\ YC_bC_r} \xrightarrow{(2)\ \text{Subsampling}} \xrightarrow{(3)\ \text{Blocks } 8\times8} \xrightarrow{(4)\ \text{DCT}} \xrightarrow{(5)\ \text{Quantization}} \xrightarrow{(6)\ \text{Entropy Coding}}$$

Table 5.7 details the analytical function and the perceptual rationale that justifies each of these stages.

Table 5.7: Stages of the JPEG compression pipeline and their respective design rationales.

Stage	Operation	Perceptual and Statistical Rationale
1	Conversion $RGB \to YC_bC_r$	Separates luminance (Y) from chrominance (C_b, C_r). The human visual system (HVS) exhibits greater sensitivity to variations in brightness than to color.
2	Chrominance subsampling (e.g., 4:2:0)	Reduces the spatial resolution of the color channels by half, discarding redundant data with negligible visual impact.

Stage	Operation	Perceptual and Statistical Rationale
3–4	Centering and application of the 8×8 DCT	Translates the pixels to the range $[-128, 127]$ and compacts the spectral energy of the block into the low-frequency coefficients.
5	Selective linear quantization	Divides each coefficient $C(u, v)$ by the corresponding element of the matrix $Q(u, v)$, applying integer rounding. This constitutes the primary source of lossy compression.
6	Zigzag scanning and coding	Orders the quantized coefficients to maximize consecutive null sequences, optimizing run-length encoding (RLE) and Huffman coding.

The **quantization matrix** $Q(u, v)$ is the central mechanism for controlling the trade-off between compression ratio and visual quality. In the practical algorithm of Figure 5.25, the quality factor specified by the user (on a scale from 1 to 100) is converted into a scalar that parameterizes the severity of the matrix Q. Reduced quality values expand the divisors of $Q(u, v)$, forcing the massive truncation of AC coefficients to zero. When this elimination is excessive, the discontinuity at the boundaries of adjacent blocks is not attenuated during reconstruction, giving rise to the so-called **blocking artifacts**.

The Logic of Zigzag Scanning

The efficiency of the entropy coder subsequent to quantization depends directly on the ordering of the data. Since the DCT concentrates the essential energy at the upper-left vertex of the matrix (low frequencies) and pushes the null coefficients toward the opposite ends, a linear reading by rows or columns would fragment the sequences of zeros.

Zigzag ordering solves this limitation by traversing the matrix diagonally in increasing order of spatial frequency. This mapping groups the significant coefficients at the beginning of the vector and concentrates the null coefficients into a single continuous sequence at the end of the arrangement, allowing the RLE algorithm to encode large blocks of data compactly and efficiently.

What is RLE?

RLE (*Run-Length Encoding*) is a lossless compression technique that encodes consecutive sequences of identical values — especially **zeros** — as a pair (count, value). In JPEG, after zigzag scanning, the quantized coefficients are organized so that the zeros are concentrated at the end of the vector. RLE then compresses this long run of zeros with extreme efficiency, optimizing the storage and transmission of the compressed image.

```
import numpy as np
import cv2
from scipy.fft import dct, idct

#   Safe Loading of the Camera Image (skimage)
try:
    from skimage import data
    img_gray = data.camera()
```

```python
except ImportError:
    import subprocess
    subprocess.run(["pip", "install", "scikit-image", "-q"])
    from skimage import data
    img_gray = data.camera()

# Slightly resizes to 256x256 to keep the pattern and speed of the previous tests
img_gray = cv2.resize(img_gray, (256, 256))

#   Luminance quantization table (standard 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 dct2(bloco):
    """2D orthogonal DCT-II (separable)."""
    return dct(dct(bloco.T, norm='ortho').T, norm='ortho')

def idct2(coefs):
    """2D orthogonal IDCT-II."""
    return idct(idct(coefs.T, norm='ortho').T, norm='ortho')

def jpeg_compress_block(bloco, Q_table):
    """DCT → quantization → dequantization → IDCT in 8×8 block."""
    C  = dct2(bloco.astype(np.float64) - 128)
    Cq = np.round(C / Q_table) * Q_table    # quantizes and dequantizes
    return np.clip(idct2(Cq) + 128, 0, 255)

def jpeg_quality_compress(img, qualidade=50):
    """Simplified JPEG: compresses entire image by 8×8 blocks."""
    if qualidade < 50:
        escala = 5000 / qualidade
    else:
        escala = 200 - 2 * qualidade
    # Fixed from 'scala' to 'escala'
    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)

#   Comparison of quality factors
qualidades = [10, 25, 50, 75, 90]
imgs_jpeg  = [img_gray]
titles_jpeg = ["Original\n(Cameraman)"]

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}\nPSNR={psnr:.1f}dB")
```

```
mm.show(imgs_jpeg, titles=titles_jpeg, cols=3, figsize=(14, 10))
```

Figure 5.25: *Pipeline* simplified JPEG applied to the classic *Cameraman* image: block DCT 8×8, quantization with different quality factors and reconstruction via IDCT. Blocking artifacts become visually evident at reduced quality factors ($Q = 10$ and $Q = 25$).

5.7.6 Interactive Simulator: DCT Quantization

The simulator in Figure 5.26 allows exploring the impact of the quantization process on an 8×8 block extracted from a real image, synthesizing in real time the following components:

- **Original and reconstructed block:** Direct representation of pixels in the spatial domain in grayscale [0, 255].
- **DCT coefficients:** Energy distribution mapped logarithmically in a chromatic gradient, highlighting the concentration of intensity at the upper left vertex (low frequencies).
- **Quantized coefficients:** Display of integer values resulting from division by the matrix $Q(u, v)$, visually making explicit the mass emergence of null coefficients (in dark tones) as the quality factor is reduced.
- **Compression metrics:** Monitoring panel that quantifies the Mean Squared Error (MSE), the number of preserved coefficients, and the volume of zeros generated for entropy coding.

5.8 Comparison of Image Formats

The choice of a digital storage format directly impacts the trade-off between visual quality, file size, and computational decoding cost. The three most relevant formats for *web* architectures and visual computing

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap05/sim-05-dct.html

Figure 5.26: Interactive DCT-JPEG compression simulator: adjust the quality factor and visualize in real time the zeroed coefficients, the reconstructed block, and the quantization error.

systems are JPEG, PNG, and WebP.

5.8.1 Characteristics of the Formats

Table 5.8 synthesizes the structural properties of the main rasterized image formats.

Table 5.8: Structural comparison between the main rasterized image formats.

Characteristic	JPEG	PNG	WebP
Compression	Lossy	Lossless	Lossy and lossless.
Transparency (alpha channel)	No	Yes	Yes.
Animation support	No	Limited (APNG)	Yes.
Base algorithm	DCT + Huffman	DEFLATE (LZ77 + Huffman)	VP8 / VP8L.
Best for	Photography	Graphics, text, and icons	Universal use in a Web environment.
Worst for	Text and sharp edges	Complex photographic images	Legacy compatibility.

5.8.2 Quality Assessment Metrics

Two objective metrics are widely adopted to quantify the distortion introduced by compression processes:

Peak Signal-to-Noise Ratio (PSNR):

$$\mathrm{PSNR} = 10 \log_{10}\left(\frac{L^2}{\mathrm{MSE}}\right) \quad [\mathrm{dB}] \tag{5.10}$$

where $L = 255$ for 8-bit quantized images and MSE represents the **Mean Squared Error**. PSNR values

above 40 dB indicate excellent fidelity; between 30 dB and 40 dB represent good quality; and values below 30 dB correspond to easily perceptible visual degradations.

Structural Similarity Index (SSIM):

$$\mathrm{SSIM}(f,g) = \frac{(2\mu_f\mu_g + c_1)(2\sigma_{fg} + c_2)}{(\mu_f^2 + \mu_g^2 + c_1)(\sigma_f^2 + \sigma_g^2 + c_2)} \tag{5.11}$$

SSIM evaluates local windows of the image based on three complementary components: **luminance** (μ_f, μ_g), **contrast** (σ_f, σ_g), and **structure** (σ_{fg}), weighted by stability constants c_1 and c_2. The index ranges over the interval $[-1, 1]$, where unity represents perfect identity. Unlike PSNR, SSIM considers the spatial organization of errors, aligning with the perception of the human visual system (HVS).

> **PSNR vs SSIM: Application of Perceptual Metrics**
>
> PSNR has a simple mathematical formulation and low computational cost; however, it tends to overestimate quality in images with localized distortions or underestimate it in global brightness variations tolerated by the observer. SSIM models biological perception with greater fidelity but requires greater processing effort. For rigorous analyses of codecs, it is recommended to report both statistical metrics in a complementary manner.

5.8.3 Visual Inspection: Nature of Compression Artifacts

The mathematical nature of the codec dictates the type of degradation introduced at reduced bit rates. As illustrated in Figure 5.27, aggressive DCT-based compression in the JPEG standard segments the image into rigid grids, generating **blocking artifacts**. In contrast, algorithms based on predictive coding or representations subjected to advanced spatial transforms (such as WebP and JPEG 2000) eliminate block discontinuities but introduce loss of fine texture and characteristic blurring around high-contrast edges.

```
import os
import cv2

# Ensures the directory exists and saves the compressed files
os.makedirs("imagens/comp_test", exist_ok=True)
cv2.imwrite("imagens/comp_test/camera_q10.jpg", img_gray, [cv2.IMWRITE_JPEG_QUALITY, 10])
cv2.imwrite("imagens/comp_test/camera_q10.webp", img_gray, [cv2.IMWRITE_WEBP_QUALITY, 10])

# Extraction of region of interest for artifact visualization (4x zoom)
zoom_original = cv2.resize(img_gray[120:200, 150:230], (320, 320),
                           interpolation=cv2.INTER_NEAREST)

rec_jpeg = cv2.imread("imagens/comp_test/camera_q10.jpg", cv2.IMREAD_GRAYSCALE)
zoom_jpeg = cv2.resize(rec_jpeg[120:200, 150:230], (320, 320),
                       interpolation=cv2.INTER_NEAREST)

rec_webp = cv2.imread("imagens/comp_test/camera_q10.webp", cv2.IMREAD_GRAYSCALE)
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=["Original Zoom", "JPEG Q=10 (Block Artifact)", "WebP Q=10 (Smoothing)"],
        cols=3, figsize=(14, 5))
```

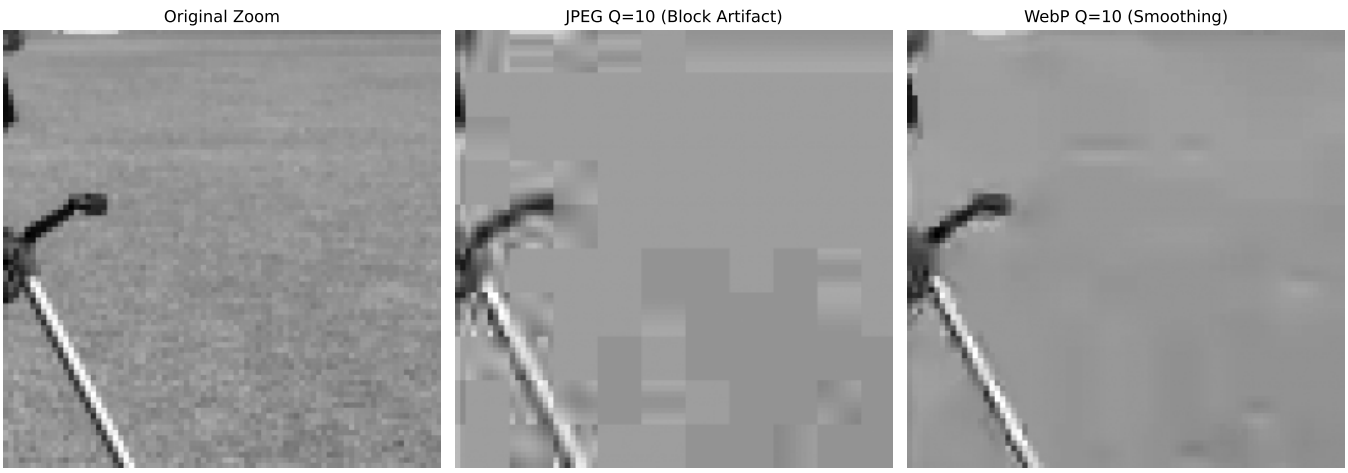


Figure 5.27: Comparative analysis of compression artifacts under reduced quality factor ($Q = 10$). On the left, the block artifact characteristic of DCT discretization in JPEG is observed. On the right, the attenuation and edge smoothing effect intrinsic to the WebP standard is evident.

5.8.4 Quantitative and Spatial Assessment of Compression

Validation of lossy compression algorithms requires an analysis that correlates storage cost with the fidelity of the reconstructed signal. This assessment is carried out complementarily through global performance curves and the local mapping of distortions induced by the coders.

5.8.4.1 Rate-Distortion Curves

Figure 5.28 presents the empirical evaluation of the JPEG and WebP pipeline through **rate-distortion curves**, which monitor compression gain (file size in KB) as a function of PSNR. The PNG format serves as an ideal baseline (PSNR $= \infty$), as its *lossless* nature prevents any degradation, although it demands a substantially larger data volume.

Analysis of the curves demonstrates the superiority and efficiency of the WebP standard over traditional JPEG: to achieve the same level of mathematical fidelity (such as the excellent quality range, where PSNR > 40 dB), the WebP encoder generates significantly smaller files. This behavior reflects the practical impact of algorithmic evolution on optimizing digital transmission and storage systems.

```
import os
import cv2
import matplotlib.pyplot as plt

# Ensures the test directory exists
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/camera_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/camera.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/camera_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.PSUB_VAL if 'cv2.PSNR' in globals() else cv2.PSNR(img_gray, rec_w),
         "KB":   os.path.getsize(path_w)/1024})

#   Rate-Distortion Curve Generation
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"]

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']:.1f} 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"\nRaw size (uncompressed): {img_gray.nbytes/1024:.0f} KB")
print(f"\n{'Format':>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}")
```

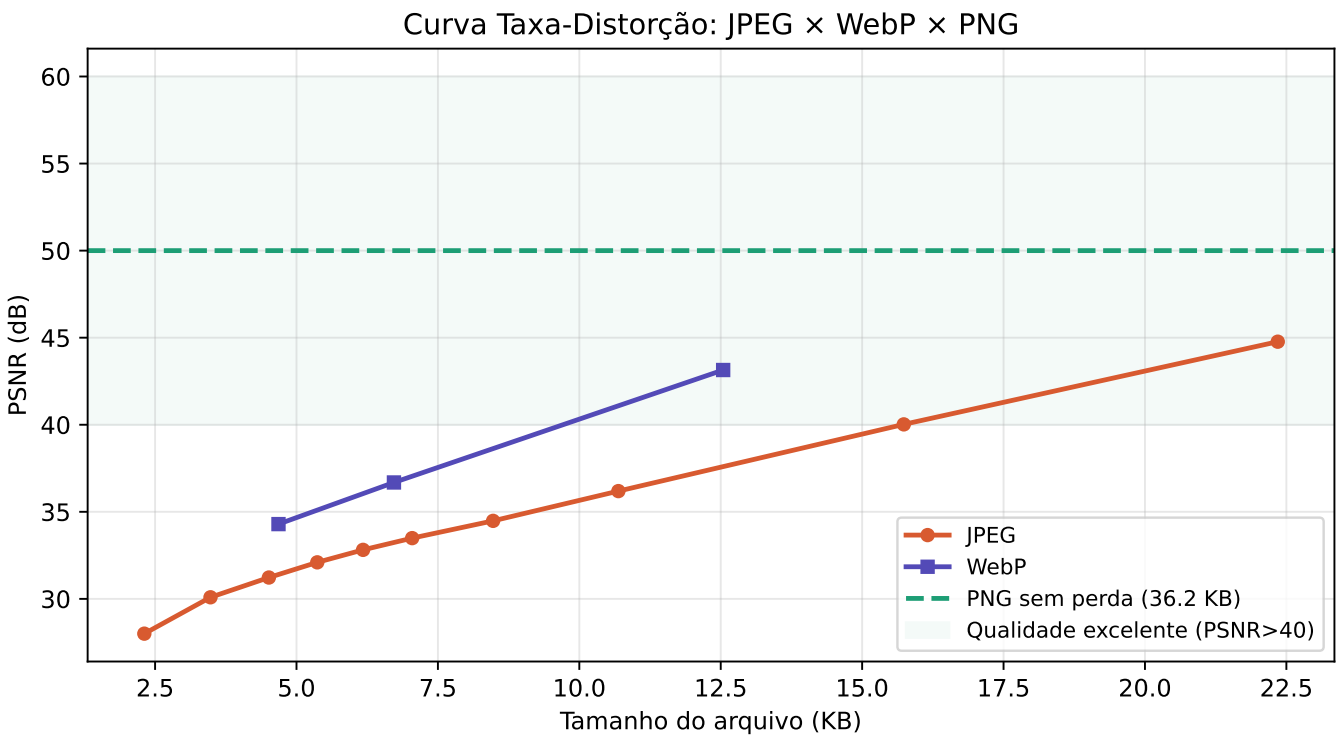


Figure 5.28: Rate-distortion curve: PSNR vs file size for JPEG, WebP, and PNG applied to the *Cameraman* image.

```
Raw size (uncompressed): 64 KB

  Format  Qual.      KB   PSNR (dB)
--------------------------------------
    JPEG     10     2.3       28.00
    JPEG     20     3.5       30.09
    JPEG     30     4.5       31.23
    JPEG     40     5.4       32.10
    JPEG     50     6.2       32.81
    JPEG     60     7.0       33.49
    JPEG     70     8.5       34.48
    JPEG     80    10.7       36.19
    JPEG     90    15.7       40.02
    JPEG     95    22.3       44.77
     PNG lossless    36.2 ϖ (lossless)
    WebP     50     4.7       34.29
    WebP     75     6.7       36.69
    WebP     90    12.5       43.14
```

Original Image Size

The *Cameraman* image (256×256 pixels in grayscale) occupies **64 KB** in raw format (uncompressed). As a reference, *lossless* PNG compresses this volume to **36.2 KB** — highlighting that lossless compression already significantly reduces storage for images with homogeneous regions. In contrast, lossy formats (JPEG and WebP) achieve even smaller sizes: JPEG at quality 95 occupies 22.3 KB (PSNR 45 dB), while WebP at quality 90 reaches 12.5 KB with equivalent PSNR, demonstrating its superior compression efficiency.

5.8.4.2 Spatial Error Mapping and Perceptual Correlation

Although PSNR provides a rapid numerical indicator, global metrics fail to discriminate how information loss is geometrically distributed across the image. Figure 5.29 addresses this limitation by associating reconstructions at different qualities with their respective absolute error and SSIM maps.

The residual maps—obtained by the normalized absolute difference between the original and compressed images—reveal the intrinsic spatial signature of each coding architecture:

- **At high qualities ($Q = 95$ to $Q = 75$):** Distortions are predominantly concentrated around abrupt intensity transitions (edges), resulting from spectral mirroring caused by the discarding of high frequencies. The SSIM index remains close to unity, attesting to the integrity of the original structures.
- **At aggressive qualities ($Q = 50$ to $Q = 25$):** The error assumes a regularized orthogonal mesh structure. This geometric pattern highlights the emergence of **blocking artifacts**, indicating that severe quantization has corrupted the spatial correlation between adjacent 8×8 pixel blocks.

SSIM captures this morphological degradation far more sensitively than PSNR, penalizing the final score as structural organization and fine textures—to which the human visual system is highly responsive—are eliminated by the encoder.

```
import os
import numpy as np
import cv2

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

# Ensures the tests directory exists
os.makedirs("imagens/comp_test", exist_ok=True)

qualidades_ssim = [25, 50, 75, 95]
imgs_ssim   = [img_gray]
titles_ssim = ["Original"]

for q in qualidades_ssim:
    path = f"imagens/comp_test/camera_ssim_q{q}.jpg"

    # FORCED RECORDING: Generates and records the JPEG with current quality at the correct
    path
    img_compactada = jpeg_quality_compress(img_gray, qualidade=q)
    cv2.imwrite(path, img_compactada)

    # Safe reading of the newly recorded file
    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)

    # Normalized absolute difference to highlight the spatial structure of the error
    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}\nPSNR={psnr_v:.1f}dB | SSIM={ssim_v:.3f}",
                    f"Mapa de erro (Q={q})\n(Bordas e blocagem)"]

mm.show(imgs_ssim, titles=titles_ssim, cols=3, figsize=(14, 14))
```

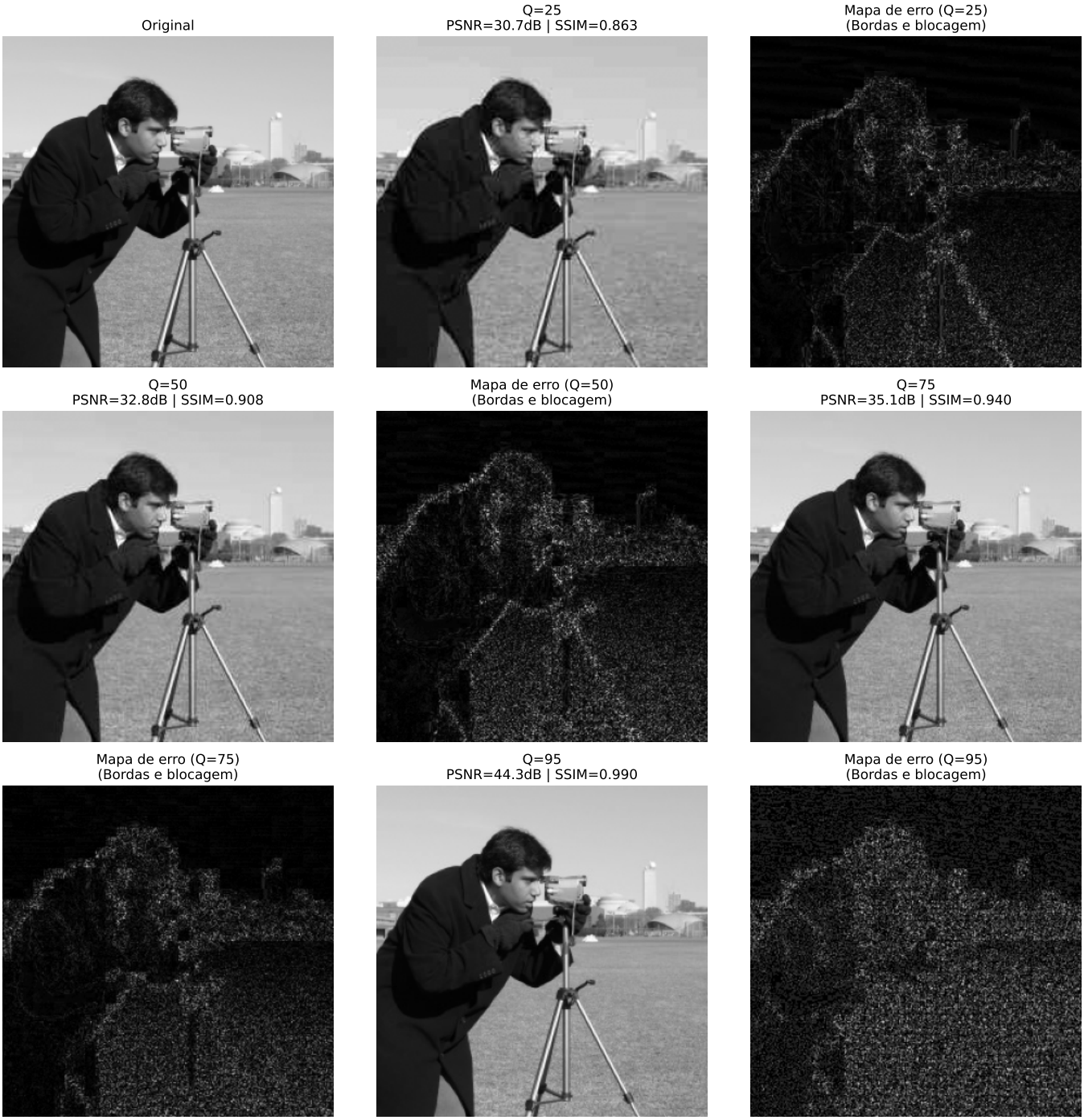


Figure 5.29: Spatial degradation analysis: reconstructed images and respective normalized absolute error maps for different JPEG quality factors.

Interpreting the Error Maps

The error maps presented were **individually normalized** (`cv2.NORM_MINMAX`) to maximize visual contrast and reveal the spatial structure of distortions. This means that:

- At **Q=95**, the absolute error is on the order of **0.5–1.5 gray levels** (visually imperceptible), but normalization amplifies it to black-and-white to highlight its location at edges and transitions.
- At **Q=25**, the absolute error is **10–20 times larger** (5–15 gray levels), but normalization also brings it to the same range [0, 255].

Therefore, **the intensity of white in the maps is NOT comparable across different qualities** — the maps serve only to reveal the **spatial signature** of the error (edges vs. blocks), not its magnitude. The correct magnitude is given by the PSNR and SSIM values, which clearly show that Q=95 has much smaller error than Q=25.

Synthesis — JPEG Compression

The compression process in the JPEG standard is based on the combined application of spatial, perceptual, and statistical transformations to reduce the redundancies of an image. Table 5.9 summarizes the role of each stage in the *pipeline* and its respective impact on data reduction.

Table 5.9: Synthesis of the stages of the JPEG compression *pipeline* and their respective impacts.

Stage	Analytical Operation	Gain / Compression Mechanism
YC_bC_r **Conversion**	Isolation of luminance and chrominance channels.	Models the perception of the HVS, allowing color and brightness to be treated independently.
4:2:0 Subsampling	Reduction of the spatial resolution of the color channels (C_b and C_r).	Eliminates approximately 50% of raw data with minimal visual impact.
8×8 **DCT**	Mapping from the spatial domain to the spatial frequency domain.	Energy compaction, concentrating vital information in the first coefficients.
Linear Quantization	Integer division of the coefficients by a weighting matrix $Q(u, v)$.	Main source of lossy compression; eliminates imperceptible high frequencies.
Entropy Coding	Application of RLE algorithms and Huffman coding.	Lossless statistical compression, optimized by long runs of null coefficients.

Characteristic Degradation Artifacts

Applying excessively aggressive compression rates (reduced quality factors) introduces predictable distortions in the reconstructed image, arising from the mathematical limitations of the model:

- **Blocking artifacts:** Visible geometric discontinuities at the boundaries of 8×8 pixel blocks, caused by the loss of spatial correlation after severe quantization of the AC components.
- **Ringing effect:** Ghost oscillations or "smoke-like" distortions around sharp, high-contrast edges, caused by the abrupt elimination of high-frequency harmonics necessary to reconstruct step functions.
- **Loss of fine texture:** Attenuation of high-frequency, low-contrast details (such as grass, fabrics, or porosity), causing originally textured regions to assume an excessively smooth or homogenized appearance.

5.9 Practical Application: Noise Removal via Hybrid Filtering

Bringing together the techniques consolidated throughout this chapter, a complete **image restoration** *pipeline* is presented, combining spectral analysis in the frequency domain with adaptive filtering in the spatial domain. The goal is to attenuate mixed noise (composed of Gaussian degradation and periodic interference) while preserving the structural details of the original image as much as possible.

$$\text{Noisy Image} \xrightarrow{\text{FFT2}} \xrightarrow{\text{Gaussian Notch Filter}} \xrightarrow{\text{IFFT2}} \xrightarrow{\text{Bilateral Filter}} \text{Restored Image}$$

Complementary Evaluation: PSNR vs. SSIM

The pair of statistical metrics PSNR and SSIM provides a complementary qualitative and morphological evaluation of the restoration process:

- **PSNR:** Uniformly penalizes the pixel-by-pixel mean squared error.
- **SSIM:** Evaluates the preservation of perceptually relevant local structures (luminance, contrast, and edges).

In practice, there is an analytical trade-off between **noise reduction** and **detail preservation**: excessively aggressive spatial filters attenuate high-frequency noise well but degrade fine textures and smooth sharp edges — which **simultaneously reduces** both the PSNR and the SSIM relative to the original image. The challenge of filter design is to find the balance point that maximizes both metrics, ensuring a faithful and visually pleasing restoration.

5.9.1 Performance Analysis and Chapter Conclusion

The numerical and visual results generated by Figure 5.30 demonstrate the practical relevance of combining different processing domains. The simultaneous insertion of periodic and stochastic noise corrupts the morphological properties of the signal, severely reducing the similarity indices and the signal-to-noise ratio of the reference image.

The isolation and suppression of harmonic peaks in the frequency domain through the *notch* mask remove the sinusoidal interference fringes spread across the two-dimensional space. As evidenced in the printed data of Figure 5.30, this surgical filtering promotes an immediate and substantial leap in the PSNR metric. However, high-frequency Gaussian noise remains uniformly active in the spectrum, requiring a complementary approach.

The final restoration is consolidated in the spatial domain with the introduction of the bilateral filter. Unlike conventional low-pass operators (such as the Gaussian or mean filter), which would indiscriminately smooth noise and structural contours, bilateral filtering computes weights based on geometric proximity and radiometric intensity difference. This adaptive behavior attenuates remaining stochastic fluctuations in smooth transition regions while preserving the sharpness of spatial edges.

The convergence of both approaches results in a **substantial and simultaneous improvement** in PSNR and SSIM relative to the noisy image — although the final values remain inferior to those of the original image (PSNR $= \infty$, SSIM $= 1.0$), due to the inevitable loss of spectral and textural information during the filtering processes. The smooth (Gaussian) attenuation of spectrum peaks avoids *ringing* artifacts, while the bilateral filter eliminates residual stochastic noise without compromising edge sharpness. The results confirm the effectiveness and practical complementarity of the frequency analysis tools presented in this chapter, demonstrating that hybrid filtering (frequency + spatial) is superior to any isolated approach for restoring images degraded by mixed noise.

```
import numpy as np
import cv2

try:
    from skimage.metrics import structural_similarity as ssim_sk
```

```
except ImportError:
    import subprocess
    subprocess.run(["pip", "install", "scikit-image", "-q"])
    from skimage.metrics import structural_similarity as ssim_sk

def suprimir_pico_gaussiano(mask, cy, cx, sigma=3.0):
    yy, xx = np.ogrid[:mask.shape[0], :mask.shape[1]]
    dist = np.sqrt((yy - cy)**2 + (xx - cx)**2)
    notch = np.exp(-dist**2 / (2 * sigma**2))
    mask *= (1 - notch)
    return mask

#   1. Construction of mixed noise
np.random.seed(42)
h_img, w_img = img_gray.shape
X2, Y2 = np.meshgrid(np.arange(w_img), np.arange(h_img))

u0, v0       = 15, 10
ruido_gauss  = np.random.normal(0, 15, img_gray.shape)
ruido_period = 30 * np.sin(2 * np.pi * (u0 * X2 / w_img + v0 * Y2 / h_img))
img_noisy    = np.clip(img_gray.astype(float) +
                       ruido_gauss + ruido_period, 0, 255).astype(np.uint8)

#   2. Spectrum and identification of peaks
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. Gaussian notch on periodic peaks
cy0, cx0       = h_img // 2, w_img // 2
mascara_notch  = np.ones((h_img, w_img), dtype=np.float64)
for dy, dx in [(+v0, +u0), (-v0, -u0), (+v0, -u0), (-v0, +u0)]:
    mascara_notch = suprimir_pico_gaussiano(mascara_notch, cy0 + dy, cx0 + dx, sigma=3.0)

img_notch = np.real(np.fft.ifft2(np.fft.ifftshift(F_n * mascara_notch)))
img_notch = np.clip(img_notch, 0, 255).astype(np.uint8)

#   4. Bilateral Filter: removal of residual Gaussian noise
img_den = cv2.bilateralFilter(img_notch, d=7, sigmaColor=25, sigmaSpace=7)

#   Calculation of Validation Metrics
psnr_n,  ssim_n  = cv2.PSNR(img_gray, img_noisy), ssim_sk(img_gray, img_noisy)
psnr_no, ssim_no = cv2.PSNR(img_gray, img_notch), ssim_sk(img_gray, img_notch)
psnr_d,  ssim_d  = cv2.PSNR(img_gray, img_den),   ssim_sk(img_gray, img_den)

print(f"{'Etapa':>20} | {'PSNR (dB)':>9} | {'SSIM':>6}")
print("-" * 42)
print(f"{'Ruidosa (gauss+per)':>20} | {psnr_n:>9.2f} | {ssim_n:>6.4f}")
print(f"{'Após notch':>20} | {psnr_no:>9.2f} | {ssim_no:>6.4f}")
print(f"{'Notch + bilateral':>20} | {psnr_d:>9.2f} | {ssim_d:>6.4f}")

mascara_vis = (mascara_notch * 255).astype(np.uint8)
mm.show(
    [img_gray, img_noisy, mag_n, mascara_vis, img_notch, img_den],
    titles=[
        "Original",
        f"Ruidosa\nPSNR={psnr_n:.1f} dB",
        "Espectro\n(picos visíveis)",
        "Máscara notch\n(gaussiana suave)",
        f"Após notch\nPSNR={psnr_no:.1f} dB",
        f"Notch + bilateral\nPSNR={psnr_d:.1f} dB  SSIM={ssim_d:.3f}"
    ],
```

```
    cols=6, figsize=(20, 4)
)
```

```
                Etapa | PSNR (dB) |   SSIM
-----------------------------------------
 Ruidosa (gauss+per) |     20.19 | 0.3486
           Após notch |     24.47 | 0.4715
   Notch + bilateral |     28.93 | 0.7426
```

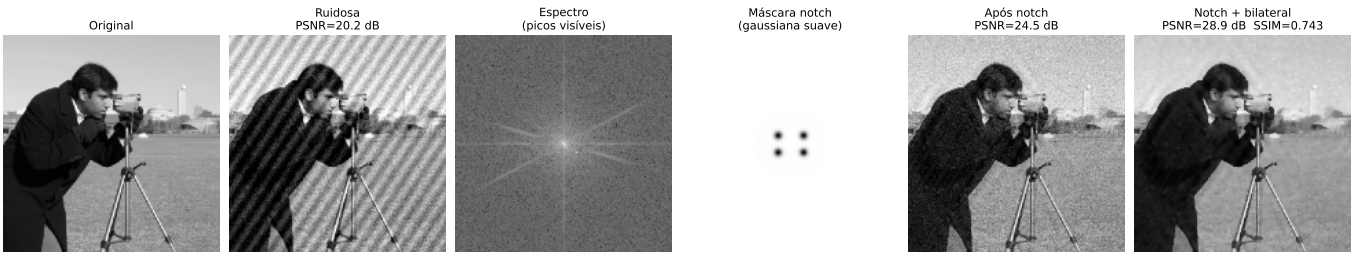


Figure 5.30: Full mixed noise removal *pipeline*: (1) addition of Gaussian and periodic noise; (2) identification of interference peaks in the frequency spectrum; (3) application of *notch* mask with smooth Gaussian attenuation; (4) post-processing via bilateral filter to eliminate residual stochastic noise.

5.10 Chapter Summary

The transition from the **spatial domain** to the **frequency domain** reveals the spectral energy distribution of the image, establishing the analytical foundation for advanced filtering, restoration, and data compression. The structural articulation of these concepts is synthesized in the conceptual map in Figure 5.31.

Essential Fundamentals

- **DFT and Visual Perception:** The spectrum decomposes the image into harmonic components. The **phase** retains the geometric intelligibility of the scene and the location of contours, while the **magnitude** dictates the distribution of contrast and global amplitudes.
- **Algorithmic Efficiency:** The Convolution Theorem enables the processing of large-scale masks in the frequency domain via FFT, reducing the asymptotic computational complexity from $O(N^2K^2)$ in space to $O(N^2 \log N)$.
- **Ringing Phenomenon:** Abrupt cuts in the spectrum (Ideal Filters) generate unwanted spatial oscillations (Gibbs phenomenon). Smooth attenuation through **Butterworth** or **Gaussian** filters eliminates these discontinuities.
- **Multiresolution Analysis via Wavelets:** Overcoming the purely global nature of Fourier, the DWT captures frequency and spatial location simultaneously, underpinning the JPEG 2000 standard and supporting hierarchical representations analogous to feature extraction in Convolutional Neural Networks (CNNs).
- **Perceptual Compression (DCT):** The JPEG pipeline exploits the contrast limitations of the human visual system at high spatial frequencies. The DCT isolates the energy of 8×8 blocks, allowing quantization to discard AC coefficients of fine details without severe perceptual loss.

Next Steps: Chapter 6 inaugurates Part II of the work, applying image processing tools to solve real industrial inspection problems. Techniques for **segmentation and shape analysis** will be explored for the automatic detection of flaws in production lines—from identifying surface defects on parts to reading *QRCode* tags on exams, consolidating the bridge between the theory presented in Part I and the practical demands of computer vision.

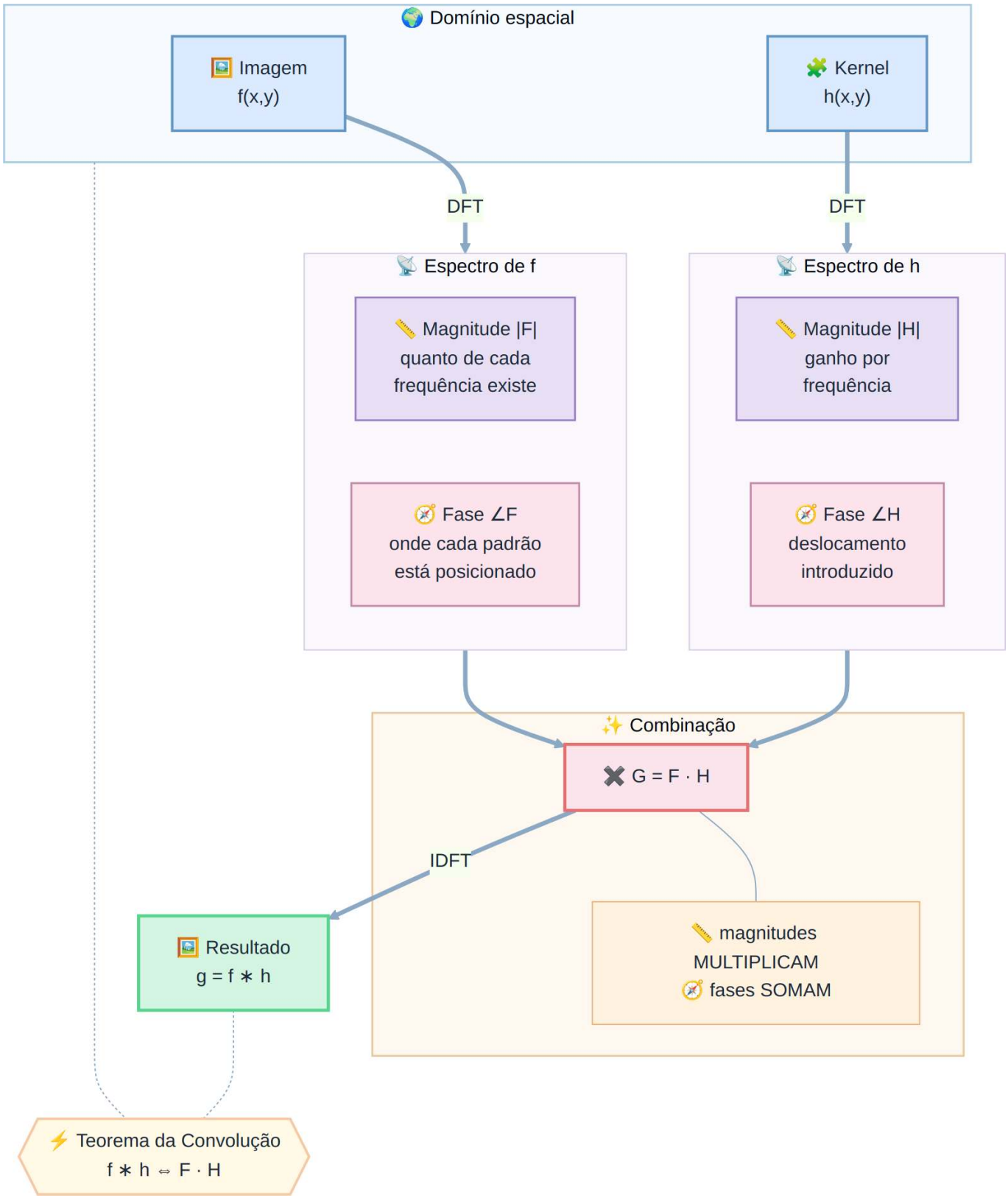


Figure 5.31: Conceptual map of transformations and properties in the frequency domain.

5.11 🤖 Using Gemini Notebook as a Complementary Tutor

In this edition, the **Gemini Notebook** platform is encouraged as a complementary learning tool — **not as a substitute** for careful reading, problem-solving, or hands-on experimentation. Based on artificial intelligence architectures, the system uses exclusively the instructional material and documents provided by the author as its knowledge base, ensuring that the generated responses are conceptually aligned with the course content and the pedagogical approach adopted throughout this work.

Accessing the Intelligent Tutor

🚀 ACCESS Gemini Notebook: CHAPTER 05

🌐 **Language and Programming Language**

The project for this chapter in Gemini Notebook was built using only the **Portuguese** text and the **Python** code examples. If you are studying from the English or French edition, or following the C++ track, the tutor's responses may not correspond exactly to the version you are reading.

Guidelines on AI-Generated Content

Although artificial intelligence tools are efficient allies in the learning and review process, the generated content is subject to inconsistencies or technical inaccuracies. Therefore, systematic consultation of textbooks, scientific articles, and indexed academic sources is essential for the rigorous validation of information. The execution and modification of the practical Python examples provided in this chapter are strongly recommended as the primary method of experimental verification of the results.

5.12 Exercise List

1. **(10%) Direct Implementation of the 2D DFT:** Analytically implement the 2D Discrete Fourier Transform (DFT) without the aid of native library functions (such as `np.fft.fft2`), using strictly the mathematical formulation defined in Equation 5.1 for a matrix of dimensions 16×16. Perform numerical validation by comparing the generated coefficients with the results of the `np.fft.fft2` function, ensuring that the maximum absolute deviation is less than 10^{-8}. Measure the execution times of both methods and provide a theoretical justification for the observed disparity in terms of asymptotic complexity.

2. **(15%) Periodic Noise Suppression:** Add sinusoidal interferences with spatial frequencies $(u_0, v_0) \in \{(5, 10), (20, 5), (30, 30)\}$ to the *Cameraman* test image. For each degradation scenario, design a specific notch filtering mask in the frequency domain to isolate and attenuate the unwanted harmonic peaks. Quantitatively evaluate the effectiveness of the restoration process by computing the PSNR and SSIM metrics. Discuss analytically the trade-off between sinusoidal noise attenuation and the undesirable attenuation of legitimate structural features of the image.

3. **(15%) Comparative Analysis of Low-Pass Operators:** Conduct a comparative study among the Ideal, Gaussian, and Butterworth low-pass filters (with harmonic orders $n = 1, 2, 4$), parameterized with cutoff frequencies $D_0 = 20, 40, 60$ pixels. For each structural combination, compute the PSNR and SSIM indices of the resulting image against the original reference signal. Organize the quantitative data in a structured table and plot the one-dimensional graphs of the corresponding transfer functions along the horizontal profile $H(u, 0)$.

4. **(15%) Haar Multiresolution Filter Bank:** Develop a script to manually perform the first-level 2D discrete wavelet decomposition using the Haar family. The algorithm must compute the corresponding low-pass (h) and high-pass (g) filter coefficients, applying them separably over the rows and columns of the matrix, followed by the decimation operation (spatial subsampling by a factor of 2). Numerically validate the accuracy of your implementation by comparing the obtained subbands with the output of

the `pywt.dwt2(img, 'haar')` function.

5. **(15%) Sparse Compression by Wavelet Thresholding:** Apply the hard thresholding filtering technique to the detail coefficients of the wavelet decomposition, adopting the numerical thresholds $T \in \{5, 10, 20, 40, 80\}$ for the Haar, Daubechies (`db4`), and Symlets (`sym4`) families. After performing the synthesis process through the inverse transform (`pywt.waverec2`), compute the PSNR and SSIM values of each reconstructed image. Identify and justify which combination of wavelet family and threshold T maximizes structural similarity.

6. **(15%) Construction of a Simplified JPEG Encoder:** Implement the complete data compression pipeline simulating the JPEG standard. The workflow must encompass: spatial conversion $RGB \rightarrow YC_bC_r$, chroma subsampling at a 4:2:0 ratio, segmentation of luminance into disjoint blocks of 8×8 pixels, application of the orthogonal 2D DCT-II, and linear quantization based on the normalized luminance matrix scaled by desired quality factors. Perform the inverse decoding and quantitatively compare the reconstructions with the files generated by the `cv2.imencode` function for quality factors of 20, 50, and 80.

7. **(15%) Perceptual Analysis on Heterogeneous Content:** Develop a synthetic image composed of three distinct regions with contrasting spectral characteristics: a complex photographic texture (representing high stochastic frequencies), a vectorized text area with sharp edges (representing pure step transitions), and a continuous linear gradient (representing homogeneous low frequencies). Submit this mixed image to compression processes under the JPEG, PNG, and WebP formats. Evaluate and interpret the results by correlating the final file size on disk with the obtained PSNR and SSIM metrics, justifying which format exhibits the best performance for signals of a heterogeneous nature and why this advantage occurs in terms of energy compaction and perceptual preservation.

Chapter References

The theoretical foundation and analytical development of the concepts addressed in this chapter are based on the following reference works:

- **Gonzalez (2018)** — Classical formulations of 2D Discrete Fourier Transforms (DFT), design of analytical filters in the frequency domain, Discrete Cosine Transform (DCT), and fundamental principles of image compression systems.
- **Oppenheim (2010)** — Formal theory of signals and systems applied in the discrete domain, covering the mathematical properties of the DFT and the analytical modeling of the Convolution Theorem.
- **Mallat (1999)** — Mathematical foundation of wavelet theory, formalization of multiresolution analysis (MRA), and architecture of dyadic filter banks.
- **Wallace (1991)** — Original specification and engineering aspects of the ISO/IEC JPEG compression standard, with emphasis on psychovisual criteria for the design of DCT quantization matrices.
- **Szeliski (2022)** — Computational modeling and characterization of modern fidelity and perceptual quality metrics (PSNR and SSIM), as well as the comparative analysis of high-performance rasterized image formats.

Run Colab Open GitHub

5.13 Chapter 5 - Exercises Proposed (EPs)

Objective: Apply the concepts presented in Chapter 5 by solving the proposed exercises.

5.14 Instructions

1. Read the statements of each exercise carefully.
2. Implement the solutions in the provided code cells.
3. Run the tests to verify your answers.
4. When finished, save and submit the notebook as directed by your instructor.

5.15 Exercise 1

Statement: Write a program that reads an RGB image and converts it to grayscale using the weighted average method. Display the original and converted images side by side.

Hint: Use `cv2.cvtColor` with `cv2.COLOR_BGR2GRAY` or implement the formula manually.

5.16 Exercise 2

Statement: Implement a function to apply a **Gaussian blur** to an image. Compare the result with the original using different kernel sizes (e.g., 3x3, 5x5, 7x7).

Hint: Use `cv2.GaussianBlur` with adjustable `ksize` and `sigmaX`.

5.17 Exercise 3

Statement: Perform edge detection using the **Canny** algorithm. Adjust the thresholds to observe their effect on the detected edges.

Hint: Use `cv2.Canny` with `threshold1` and `threshold2` parameters.

5.18 Exercise 4

Statement: Use morphological operations (erosion and dilation) to remove noise from a binary image. Show the results after each operation.

Hint: Use `cv2.erode` and `cv2.dilate` with a structuring element (e.g., `cv2.getStructuringElement`).

5.19 Exercise 5

Statement: Develop a simple **image segmentation** algorithm by thresholding (Otsu's method). Display the histogram and the resulting binary image.

Hint: Use `cv2.threshold` with `cv2.THRESH_BINARY + cv2.THRESH_OTSU`.

5.20 Submission

After completing all exercises, ensure that the notebook runs without errors. Include comments in your code explaining the main steps. Submit the file according to your course's guidelines.

End of Chapter 5 Proposed Exercises.

5.21 💻 Practical Part with Programming Exercises

The present list of programming exercises (EP) consolidates the theoretical formulations presented throughout Chapter 5 — Transforms and Compression — through an applied practical track. The exercises are structured around matrices of reduced dimensions, enabling analytical validation and manual inspection of each coefficient, while maintaining the methodological consistency adopted in previous chapters.

The sequencing of the exercises strictly reproduces the conceptual flow of the chapter: it begins with the explicit implementation of the Discrete Fourier Transform (DFT) from its fundamental mathematical definition; it proceeds to the design of low-pass filters and *notch* masks in the frequency domain; it applies coefficient quantization (the core of lossy compression); and it concludes with the integration of these stages in the construction of a simplified JPEG compression *pipeline* and in the perceptual analysis of image formats.

❗ Guidelines for Solving the Programming Exercises

In all exercises in this chapter, the coordinates of the **spectrum center** (the origin of spatial frequencies after the application of the `fftshift` shift) must be determined using integer division. For a matrix with L rows and C columns, the zero-frequency component is located at the position:

$$(c_y, c_x) = \left(\left\lfloor \frac{L}{2} \right\rfloor, \left\lfloor \frac{C}{2} \right\rfloor \right)$$

This convention is strictly identical to that adopted by the `np.fft.fftshift` function. Furthermore, in all stages that require discretization or numerical rounding (whether in the quantization of AC coefficients or in the final reconstruction of pixels), the standard rounding to the nearest integer (*round half away from zero*) must be employed, mitigating ambiguities in values with a fraction exactly equal to 0.5.

🎯 Objective of this Notebook

The notebook enables the development, validation, organization, and testing of solutions for **Programming Exercises (PEs)** in interactive environments, such as Colab, using the same test cases as Moodle, copying them there only when recording the official grade.

Download

Download `morph.py` and `testsuite.py` by running the cell below:

```
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup(testsuite=True)
from morph import mm
from testsuite import TestSuite
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Running the Tests

To evaluate the tests, run `TestSuite("EP05_01.extensão").run()` in a new cell, replacing the extension with the one used for your language (`.py`, `.java`, `.c`, `.cpp`, `.js`, or `.r`). The system downloads the test cases from GitHub, runs the program, and calculates the grade automatically.

To test Python code directly, without saving a file, use `run_code(codigo)` passing the code as a *string* in a variable named `codigo`:

```python
codigo = """
from morph import mm
# ... your code here ...
"""
TestSuite("EP05_01").run_code(codigo)
```

5.21.1 EP05_01 🟢 Ideal Low-Pass Filter by Distance in the Spectrum

In an **old document scanner**, the sensor captures crumpled paper and fiber texture along with the text — high-frequency noise that "pollutes" the spectrum at the edges. The maintenance technician has no access to the original image, only to the **magnitude spectrum already computed** by the scanner's software. Their job is simple and surgical: keep only the **central circle** of low frequencies (the global structure of the document) and erase everything outside the radius D_0, eliminating the fine texture without even needing to touch the spatial image.

This is the **Ideal Low-Pass Filter (LPFI)**: the most direct spectral operation in the chapter, but also the one that best reveals the anatomy of a centered spectrum.

5.21.1.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns) from the magnitude spectrum — already provided **centered** (equivalent to the output of `np.fft.fftshift`).
2. **Cutoff frequency:** Read the integer D_0.
3. **Data:** Read the integer values of the magnitude matrix, row by row.
4. **Spectrum center:** Compute $(c_y, c_x) = (L \,//\, 2,\ C \,//\, 2)$.
5. **Distance:** For each position (u, v), compute

$$D(u,v) = \sqrt{(u - c_y)^2 + (v - c_x)^2}$$

6. **Ideal mask:** Apply

$$H(u,v) = \begin{cases} 1, & D(u,v) \le D_0 \\ 0, & D(u,v) > D_0 \end{cases}$$

7. **Filtering:** The output value is $\text{mag}'(u,v) = \text{mag}(u,v) \cdot H(u,v)$.
8. **Output:** Display the filtered matrix with dimensions $L \times C$.

5.21.1.2 📌 Computational Constraints

- **Non-strict comparison:** the criterion uses $D(u,v) \le D_0$ (the boundary belongs to the filter, i.e., it is kept).
- **Type:** all input and output values are integers; the distance is computed in floating point only internally.
- **No magnitude rounding:** since the input is already integer and the mask is binary (0 or 1), the output never requires rounding.

5.21.1.3 🧠 Theoretical Background

Region	Distance to center	Filter effect
Center ($D \leq D_0$)	Low frequencies	Preserved — global structure maintained
Edges ($D > D_0$)	High frequencies	Zeroed — texture and noise removed
Small D_0	—	Reconstructed image would be very blurry
Large D_0	—	Little filtering; almost all energy preserved

5.21.1.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer D_0.
- Following lines: Integer elements of the magnitude matrix (centered).

Output:

- Filtered matrix with L rows and C columns, separated by spaces.

5.21.1.5 📌 Examples

Input	Output	Observation
3 3 1 10 20 30 40 50 60 70 80 90	0 20 0 40 50 60 0 80 0	Center $(1, 1)$. Corners have $D = \sqrt{2} \approx 1.41 > 1$, hence they are zeroed; orthogonal neighbors have $D = 1 \leq 1$ and are kept.
1 3 0 5 9 7	0 9 0	$L = 1, C = 3$: center at $(0, 1)$. Only the central position itself ($D = 0$) survives $D_0 = 0$.

```
%%writefile EP05_01.py
# Python code
```

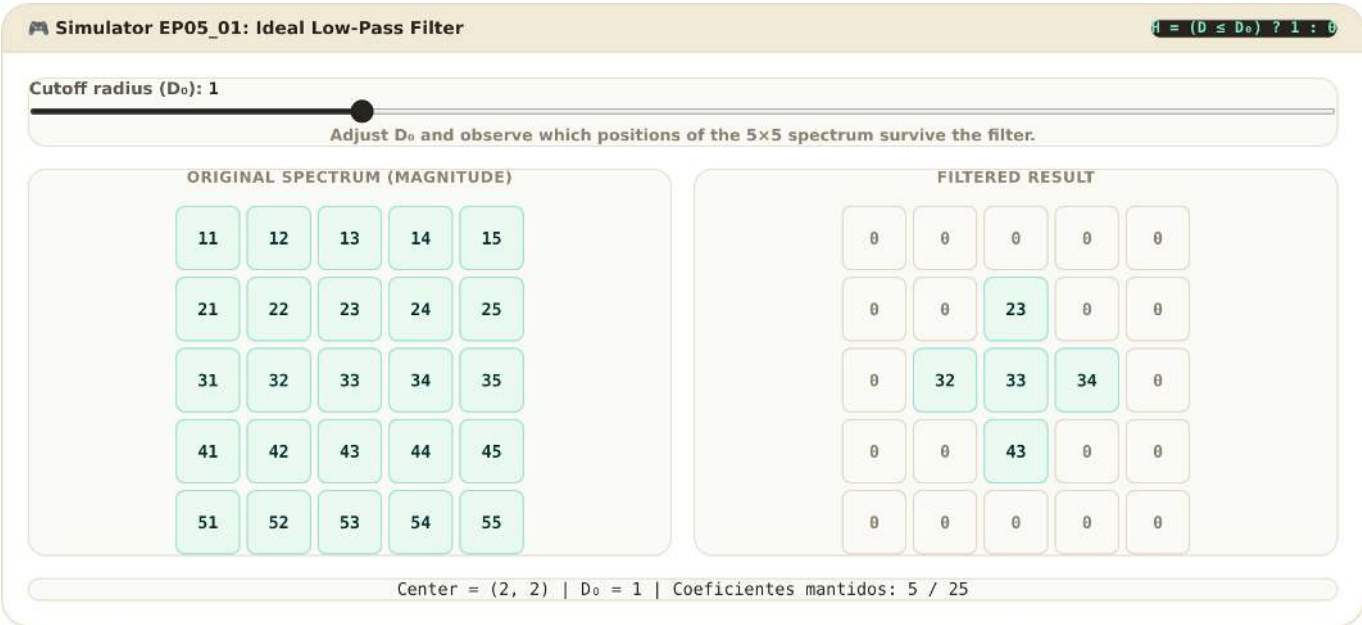
```
Overwriting EP05_01.py
```

```
TestSuite("EP05_01.py").run()
```

```
<IPython.core.display.HTML object>
```

5.21.2 EP05_02 🟡 *Notch* Filter: Removing Periodic Peaks

An **industrial inspection** camera captures images of circuit boards, but the production line's power supply introduces a **periodic electrical interference** — a stripe pattern almost imperceptible to the naked eye,



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap05/sim-ep0501.html

Figure 5.32: EP05_01 Simulator: Ideal Low-Pass Filter in the Spectrum

yet visible in the Fourier spectrum as **pairs of bright peaks** symmetrically positioned around the center. The computer vision team cannot redo the capture: they must **surgically locate and erase** these peak pairs in the spectrum, preserving all other useful image information.

This is the role of the **notch reject filter**: unlike a low-pass filter (which affects a continuous region), it targets **specific points and their symmetric counterparts**, leaving the rest of the spectrum untouched.

5.21.2.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns) of the centered magnitude spectrum.
2. **Data:** Read the integer values of the magnitude matrix, row by row.
3. **Peaks:** Read the integer K (number of peak pairs to remove).
4. **For each of the K peaks:** read three integers Δv, Δu, r — vertical offset, horizontal offset, and notch radius.
5. **Spectrum center:** $(c_y, c_x) = (L \,//\, 2,\ C \,//\, 2)$.
6. **Symmetric suppression:** for each peak, zero out **all** positions (u, v) such that the distance to the point $(c_y + \Delta v,\ c_x + \Delta u)$ is $\leq r$, **and also** all positions with distance $\leq r$ to the symmetric point $(c_y - \Delta v,\ c_x - \Delta u)$.
7. **Output:** Display the resulting matrix with dimensions $L \times C$.

5.21.2.2 📌 Computational Constraints

- **Mandatory symmetry:** each reported peak generates **two** zeroed disks (the point and its symmetric counterpart relative to the center) — forgetting the symmetric point is the most common mistake.
- **Overlap:** if two disks overlap, the position remains zeroed (there is no "addition" or restoration).
- **Non-strict comparison:** a position is zeroed if distance $\leq r$.
- **Reading order:** the K peaks must be processed in the order they appear in the input, but the final result is independent of order (zeroing operations are commutative).

5.21.2.3 🧠 Theoretical Foundation

Concept	Role in the notch filter
Peak at $(\Delta v, \Delta u)$	Frequency of the periodic interference visually detected in the spectrum
Symmetric point $(-\Delta v, -\Delta u)$	Every DFT of a real signal is Hermitian: peaks always appear in pairs symmetric about the center
Radius r	Controls the "width" of rejection — a large r removes more energy around the peak, but also useful information

5.21.2.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Following lines: Integer elements of the magnitude matrix (centered), L lines.
- Next line: Integer K.
- Following K lines: three integers Δv, Δu, r (space-separated).

Output:

- The resulting matrix in L lines and C columns, space-separated.

5.21.2.5 📌 Examples

Input	Output	Observation
5 5 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 1 1 1 0	1 2 3 4 5 6 0 8 9 10 11 12 13 14 15 16 17 18 0 20 21 22 23 24 25	Center $(c_y, c_x) = (2, 2)$. Reported peak $(\Delta v, \Delta u) = (1, 1)$ generates the point $(3, 3)$ (value 19) and its symmetric counterpart $(1, 1)$ (value 7), both zeroed with $r = 0$ (only the exact points).

```
%%writefile EP05_02.py
# Python code
```

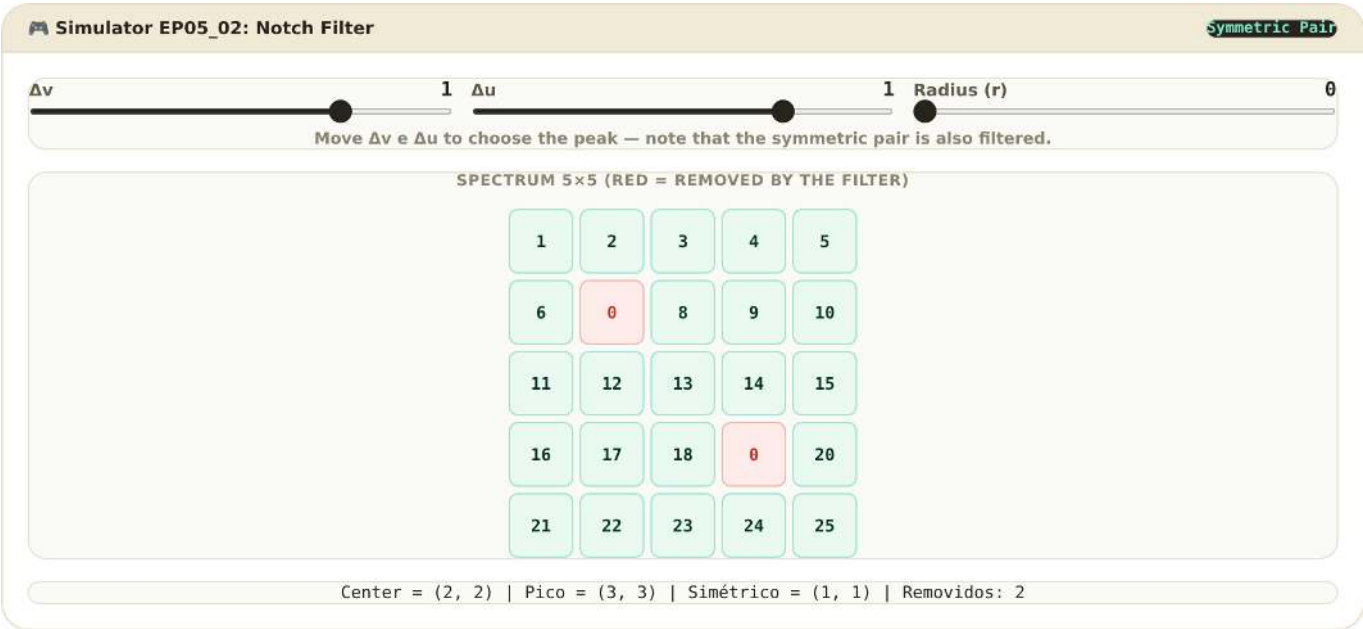
```
Overwriting EP05_02.py
```

```
TestSuite("EP05_02.py").run()
```

```
<IPython.core.display.HTML object>
```

5.21.3 EP05_03 🟠 DCT Quantization: The True Source of Compression

A **photo gallery** application needs to reduce the size of thousands of images before *uploading* them to the cloud, without recoding everything from scratch. The engineer in charge already has the **DCT coefficients** for each 4×4 block calculated (the computationally expensive step has already been done) — only the **quantization table** needs to be applied, the step that actually discards information and generates compression. High-frequency coefficients, which are less perceptible to the human eye, receive large divisors



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap05/sim-ep0502.html

Figure 5.33: EP05_02 Simulator: Notch Filter

and tend to become **zero**; low-frequency coefficients, which are more perceptible, receive small divisors and survive almost intact.

You will implement exactly this step: **quantize and dequantize** (divide, round, multiply back) — the heart of JPEG *lossy* compression.

5.21.3.1 📋 Implementation Guidelines

1. **Block size:** Read the integer N ($N \times N$ block).
2. **Coefficients:** Read the matrix C of DCT coefficients, N rows with N integers each (they may be negative).
3. **Quantization table:** Read the matrix Q, N rows with N positive integers each.
4. **Quantization:** For each position (u, v), compute the quantized index

$$\tilde{C}(u, v) = \text{round}\left(\frac{C(u, v)}{Q(u, v)}\right)$$

   using standard rounding to the nearest integer (intermediate `.5` values never occur in the test cases).
5. **Dequantization (reconstruction):** Compute

$$C'(u, v) = \tilde{C}(u, v) \times Q(u, v)$$

6. **Output:** Display the reconstructed matrix C', $N \times N$, integers.

5.21.3.2 📌 Computational Constraints

- **Complete *round-trip*:** The output is the **reconstructed** coefficient ($\tilde{C} \times Q$), not the isolated quantized index.
- **Floating-point division:** The division $C(u, v)/Q(u, v)$ must be performed in floating point before rounding — truncated integer division will produce an incorrect result.
- **Preserved sign:** Negative coefficients retain their sign after quantization and reconstruction.
- $Q(u, v) > 0$ **always:** There is no need to handle division by zero.

5.21.3.3 🧠 Theoretical Background

Coefficient	Frequency	Typical value of Q	Effect of quantization
$C(0,0)$	DC (block average)	Small	Almost always survives — dominates energy
$C(u,v)$ low $u+v$	Low frequency	Small/medium	Partially preserved
$C(u,v)$ high $u+v$	High frequency	Large	Often becomes zero — source of compression

5.21.3.4 📦 Input and Output Specification (VPL)

Input:

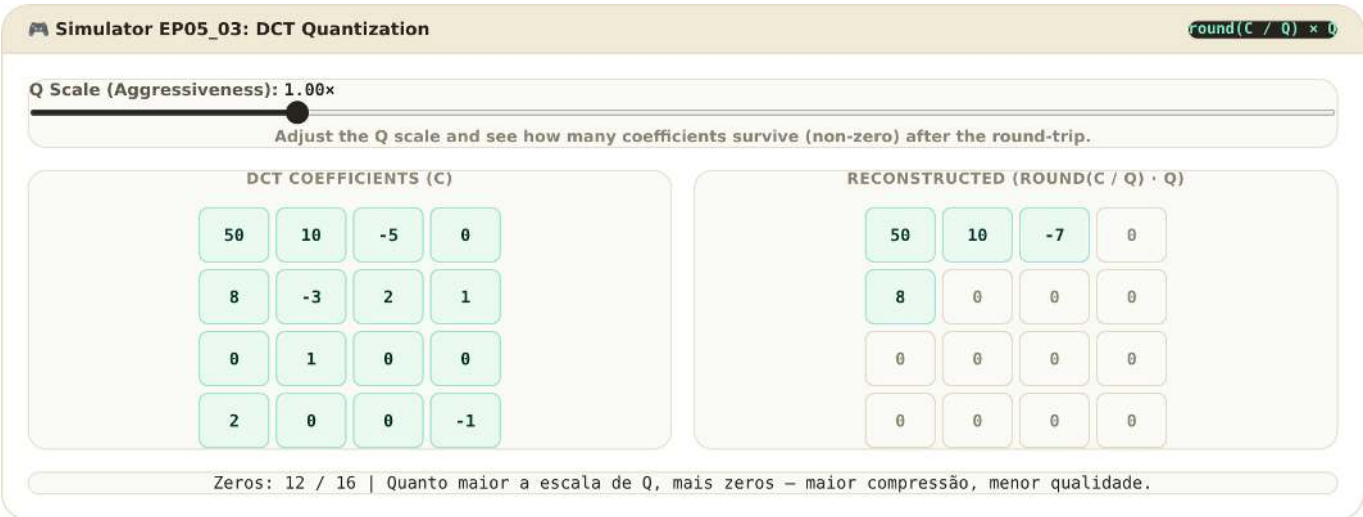
- Line 1: Integer N.
- The next N lines: matrix C (DCT coefficients, integers, may be negative).
- The next N lines: matrix Q (quantization table, positive integers).

Output:

- Reconstructed matrix C', $N \times N$, integers separated by spaces.

5.21.3.5 📌 Examples

Input	Output	Observation
4 50 10 -5 0 8 -3 2 1 0 1 0 0 2 0 0 -1 2 5 7 8 4 7 8 11 6 8 11 12 9 11 12 14	50 10 -7 0 8 0 0 0 0 0 0 0 0 0 0 0	$C(0,0) = 50/2 = 25 \to 25{\times}2 = 50$ (preserved). $C(0,2) = -5/7 \approx -0.71 \to -1 \to -1 \times 7 = -7$. Meanwhile, $C(1,1) = -3/7 \approx -0.43 \to 0$: zeroed by quantization — most of the block becomes zero, illustrating the energy compaction in the upper-left corner.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap05/sim-ep0503.html

Figure 5.34: EP05_03 Simulator: DCT Quantization (*round-trip*)

```
%%writefile EP05_03.py
# Python code
```

```
Overwriting EP05_03.py
```

```
TestSuite("EP05_03.py").run()
```

```
<IPython.core.display.HTML object>
```

5.21.4 EP05_04 🔴 Implementing the 2D DFT from the Definition

A research laboratory in **computational astronomy** received, from an old mission, a small experimental sensor whose raw data cannot be processed by modern FFT libraries — the validation environment is isolated and only allows basic arithmetic operations. The team needs to **reimplement the 2D Discrete Fourier Transform from the mathematical definition itself**, cell by cell, to later compare bit by bit with `np.fft.fft2` in another environment.

This is the most conceptual exercise on the list: there are no shortcuts. You will implement the double summation from Equation 5.1 directly, demonstrating *why* the FFT exists — and the computational cost it avoids.

5.21.4.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers M (rows) and N (columns) of the image $f(x, y)$.
2. **Data:** Read the integer values of $f(x, y)$, row by row.
3. **2D DFT:** For each frequency pair (u, v) with $u = 0, \dots, M-1$ and $v = 0, \dots, N-1$, compute

$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y)\, e^{-j2\pi\left(\frac{ux}{M} + \frac{vy}{N}\right)}$$

   using Euler's identity $e^{-j\theta} = \cos(\theta) - j\sin(\theta)$ to separate the real and imaginary parts — **do not use any ready-made FFT function**.
4. **Magnitude:** Compute $|F(u, v)| = \sqrt{\mathrm{Re}(F)^2 + \mathrm{Im}(F)^2}$ and round to the nearest integer.
5. **Output:** Display the matrix of rounded magnitudes, $M \times N$, in the same order (without `fftshift` — the DC remains at $(0, 0)$).

5.21.4.2 📌 Computational Constraints

- **Using FFT libraries is prohibited:** the implementation must compute the double summations explicitly (nested loops), even if slower.
- **No `fftshift`:** the output maintains the raw DFT convention, with the DC component at $F(0, 0)$ (upper-left corner).
- **Rounding:** the final magnitude must be rounded to the nearest integer; in the test cases there is no `.5` ambiguity.
- **Precision:** small floating-point errors (on the order of 10^{-6}) before rounding are expected and do not affect the final integer result.

5.21.4.3 🧠 Theoretical Foundation

Element	Meaning
$F(0, 0)$	DC component — sum of all pixels, $F(0, 0) = \sum f(x, y)$
Real part $\mathrm{Re}(F)$	Projection of the signal onto cosines
Imaginary part $\mathrm{Im}(F)$	Projection of the signal onto sines

Element	Meaning
Complexity of this implementation	$\mathcal{O}((MN)^2)$ — this is why the FFT, with $\mathcal{O}(MN\log(MN))$, is indispensable for real images

5.21.4.4 📦 Input and Output Specification (VPL)

Input:

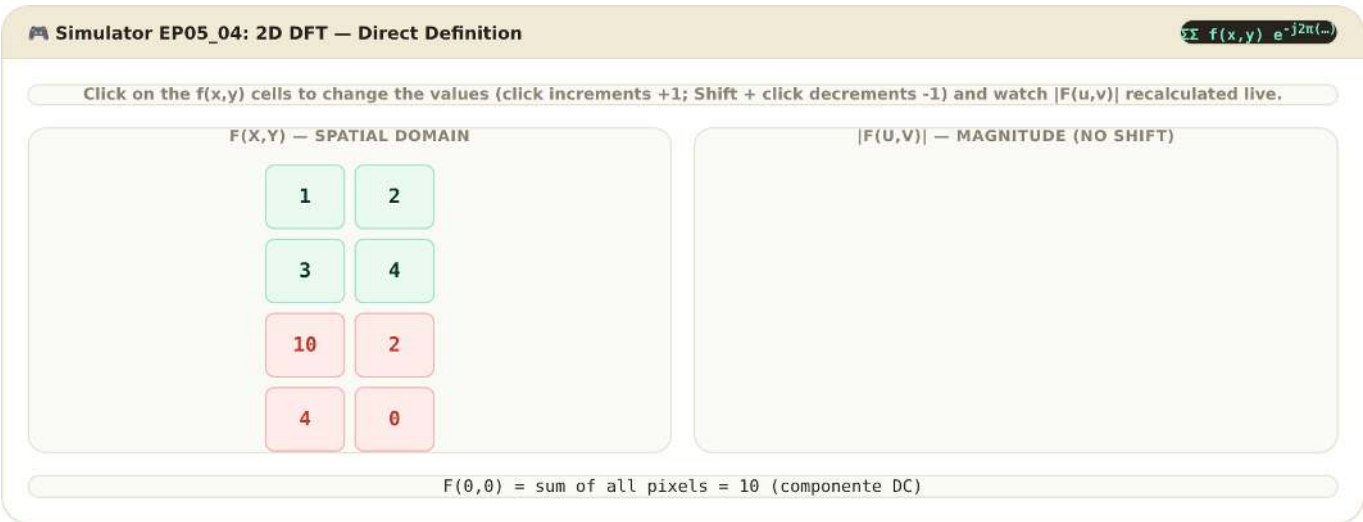
- Line 1: Integer M.
- Line 2: Integer N.
- Following lines: Integer elements of $f(x, y)$, M lines.

Output:

- Matrix of rounded magnitudes $|F(u, v)|$, $M \times N$, separated by spaces.

5.21.4.5 📌 Examples

Input	Output	Observation
2 2 1 2 3 4	10 2 4 0	$F(0,0) = 1+2+3+4 = 10$ (DC = total sum). $F(0,1) = (1-2)+(3-4) = -2 \to \lvert F\rvert = 2$. $F(1,0) = (1+2)-(3+4) = -4 \to \lvert F\rvert = 4$. $F(1,1) = (1-2)-(3-4) = 0$.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap05/sim-ep0504.html

Figure 5.35: EP05_04 Simulator: Manual 2D DFT

```
%%writefile EP05_04.py
# Python code
```

```
Overwriting EP05_04.py
```

```
TestSuite("EP05_04.py").run()
```

```
<IPython.core.display.HTML object>
```

5.21.5 EP05_05 🏆 Complete JPEG Pipeline: DCT, Quantization, and Reconstruction

You have been hired to create, from scratch, a **didactic JPEG codec** in an embedded environment, without any available image library—only basic mathematical operations. The client wants to understand exactly where quality is lost and where it is recovered, block by block. This is the final challenge of the chapter: integrate **everything** that has been studied—the orthonormal DCT-II, perceptual quantization, and IDCT-based reconstruction—into a single end-to-end *pipeline*, processing an $N \times N$ block from start to finish, exactly as the JPEG standard does internally, 8×8 pixels at a time.

5.21.5.1 📋 Implementation Guidelines

1. **Block dimension:** Read the integer N.
2. **Original block:** Read the pixel matrix $f(x, y)$, N rows with N integers in $[0, 255]$.
3. **Quantization table:** Read the matrix Q, $N \times N$ positive integers.
4. **Centering:** Subtract 128 from each pixel: $g(x, y) = f(x, y) - 128$.
5. **Orthonormal 2D DCT-II:** Compute

$$C(u, v) = \alpha(u)\,\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} g(x, y)\, \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right]$$

   with $\alpha(0) = \sqrt{1/N}$ and $\alpha(k) = \sqrt{2/N}$ for $k > 0$.
6. **Quantization:** $\tilde{C}(u, v) = \text{round}(C(u, v)/Q(u, v))$.
7. **Dequantization:** $C'(u, v) = \tilde{C}(u, v) \times Q(u, v)$.
8. **2D IDCT-II (orthonormal inverse):** Compute $g'(x, y)$ from $C'(u, v)$ using the corresponding inverse transform (same basis, summation over u, v).
9. **Centering reversal and rounding:** $f'(x, y) = \text{round}(g'(x, y) + 128)$, restricted to the interval $[0, 255]$ (*clipping*).
10. **Output:** Display the reconstructed block f', $N \times N$, integers.

5.21.5.2 📌 Computational Constraints

- ***Complete pipeline mandatory:** all six stages (center, DCT, quantize, dequantize, IDCT, revert) must be implemented—skipping quantization will not pass the tests, as the result would be identical to the original.
- ***Clipping:** reconstructed values outside $[0, 255]$ must be truncated (0 if negative, 255 if greater than 255).
- **Rounding:** both in quantization and in final pixel reconstruction, use standard rounding; the test cases avoid `.5` ambiguity.
- **Orthonormal basis:** the normalization $\alpha(u)$ and $\alpha(v)$ must be applied exactly as specified—without it, the IDCT will not reconstruct correctly.

5.21.5.3 🧠 Theoretical Foundation

Stage	Analogous in the real JPEG standard	Where quality is lost
Centering	Same—DCT assumes a signal centered at zero	No loss
DCT-II	Steps 3–4 of the *pipeline* (Table 5.7)	No loss (exact and reversible transformation)
Quantization	Step 5—division by $Q(u, v)$	**Main source of loss**—high-frequency coefficients become zero
IDCT	Final reconstruction	Reconstructs exactly the *quantized* coefficients, not the original ones

5.21.5.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer N.
- Next N lines: original block $f(x, y)$, integers in $[0, 255]$.
- Next N lines: quantization table Q, positive integers.

Output:

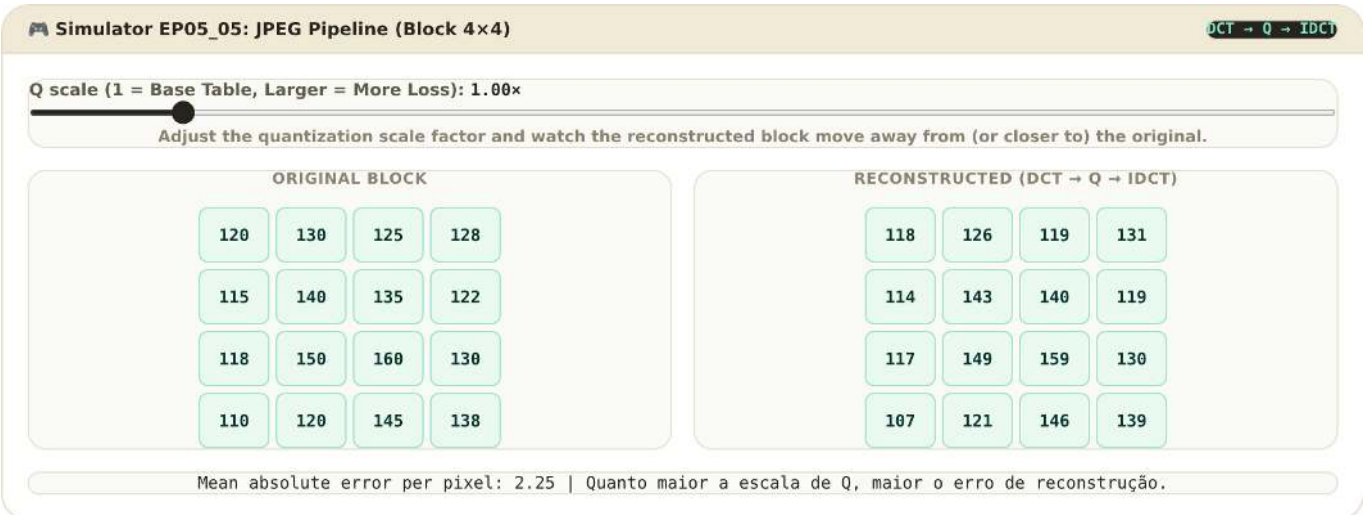
- Reconstructed block $f'(x, y)$, $N \times N$, integers in $[0, 255]$, separated by spaces.

5.21.5.5 📌 Examples

Input	Output	Observation
4 120 130 125 128 115 140 135 122 118 150 160 130 110 120 145 138 4 6 8 10 6 8 10 12 8 10 12 16 10 12 16 20	118 126 119 131 114 143 140 119 117 149 159 130 107 121 146 139	After DCT, aggressive quantization of high frequencies (large Q values in the lower-right corner) and IDCT-based reconstruction, the block remains **close** to the original but not identical—the difference is the cost of *lossy* compression.

5.21.5.6 💡 Debugging Tip

If the result does not match, check in this order: (1) the raw DCT coefficients (before quantization)—they should reconstruct the original **exactly** via IDCT if you skip steps 6–7; (2) the $\alpha(u)$ table—a common mistake is applying $\sqrt{2/N}$ also for $u = 0$; (3) the quantization rounding, which must occur **before** multiplying back by Q.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap05/sim-ep0505.html

Figure 5.36: EP05_05 Simulator: Complete JPEG pipeline in block

```
%%writefile EP05_05.py
# Python code
```

```
Overwriting EP05_05.py
```

```
TestSuite("EP05_05.py").run()
```

```
<IPython.core.display.HTML object>
```

Part II

Part II — Computer Vision

Chapter 6

Document Analysis and Industrial Inspection

Run Colab Open GitHub

In **Part I — Digital Image Processing (DIP)**, techniques for image transformation and enhancement were studied, such as morphological operations, spatial filtering, convolutions, thresholding, segmentation, and frequency-domain processing.

Part II — Computer Vision (CV) expands this scope by addressing the automatic interpretation of visual content, involving information extraction, pattern recognition, and decision-making from images.

This chapter presents this transition through two representative applications:

1. **Automated Document Analysis**, applied to the processing of forms, assessments, and other structured documents through optical mark recognition (OMR) systems;
2. **Automated Industrial Inspection**, focused on quality control and defect detection in production lines.

These applications integrate techniques for geometric structure detection, extraction of invariant descriptors, pattern recognition, and object classification, forming the foundation of various modern visual inspection and automation systems.

6.1 Chapter Objectives

By the end of this chapter, the student should be able to:

- **Evaluate the influence of preprocessing** on the quality of automatic information recognition in documents;
- **Perform optical character recognition** (OCR) to convert scanned documents into encoded text;
- **Apply natural language processing techniques**, including machine translation, to the text obtained through OCR;
- Apply techniques for **geometric document alignment and rectification** using the Hough Transform and projective transformations;
- **Implement optical mark recognition** (OMR) systems for the automated reading of assessments and forms;
- **Detect and segment regions of interest** based on morphological operations, contours, and geometric properties;
- **Decode two-dimensional markers and barcodes**, integrating Computer Vision libraries into document processing *pipelines*;
- **Develop Computer Vision *pipelines*** for automated document analysis.

This chapter marks the transition from **Digital Image Processing**, focused on image transformation, to **Computer Vision**, whose goal is to interpret visual content, extract information, and support automated analysis and decision-making processes.

6.2 Environment Setup

The examples in this chapter use libraries widely employed in CV-IP. The block below installs the necessary packages; in environments that already have them, the execution can be skipped.

```
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup()
from morph import mm

# install more dependencies beyond morph.py for this chapter
import sys, subprocess, importlib, shutil

def setup_cap06():
    """Installs system and Python-specific dependencies for Chapter 6
    (OCR, PDF reading, barcodes)."""

    # 1. System dependencies
    if 'google.colab' in sys.modules:
        print("[ENVIRONMENT] Google Colab. Setting up system dependencies...")
        subprocess.run(
            "apt-get update && apt-get install -y poppler-utils "
            "libzbar0 tesseract-ocr tesseract-ocr-por",
            shell=True, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL
        )
    elif shutil.which("tesseract") is None:
        # Local environment without tesseract: tries to install via apt-get (requires
        sudo/root)
        if shutil.which("apt-get"):
            print("[ENVIRONMENT] Local. Installing tesseract-ocr via apt-get (may ask for
            password)...")
            resultado = subprocess.run(
                "sudo apt-get update && sudo apt-get install -y tesseract-ocr
                tesseract-ocr-por",
                shell=True
            )
            if resultado.returncode != 0 or shutil.which("tesseract") is None:
                print(
                    "[WARNING] Could not install automatically. "
                    "Install manually: sudo apt install tesseract-ocr tesseract-ocr-por"
                )
        else:
            print(
                "[WARNING] tesseract not found and apt-get unavailable. "
                "Install manually before running the OCR cells."
            )

    # 2. Python dependencies (installs only missing ones)
    pkgs = {
        "cv2": "opencv-python", "skimage": "scikit-image", "numpy": "numpy",
        "pdf2image": "pdf2image", "pandas": "pandas", "tabulate": "tabulate",
        "PyPDF2": "PyPDF2", "bcrypt": "bcrypt", "pyarrow": "pyarrow",
        "pyzbar": "pyzbar", "pytesseract": "pytesseract", "deep_translator": "deep-translator"
    }
    for mod, pkg in pkgs.items():
        if importlib.util.find_spec(mod) is None:
```

```
            resultado_pip = subprocess.run([sys.executable, "-m", "pip", "install", "-q",
pkg])
            if resultado_pip.returncode != 0:
                print(f"[WARNING] Failed to install {pkg} (required for module {mod}).")


setup_cap06()

# 3. Global pipeline imports
import cv2, numpy as np, matplotlib.pyplot as plt
from skimage import io, data, color
```

```
 Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0
[ENVIRONMENT] Local. Installing tesseract-ocr via apt-get (may ask for password)...
[WARNING] Could not install automatically. Install manually: sudo apt install tesseract-ocr
tesseract-ocr-por
```

In addition to these libraries, the didactic module `morph.py` will be used, developed to simplify image reading, visualization, and processing operations throughout this book. The following code checks its availability, performs the *download* when necessary, and confirms the loaded version.

```
import os
import urllib.request

if not os.path.exists("morph.py"):
    urllib.request.urlretrieve(
        "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/morph.py",
        "morph.py",
    )

import morph
from morph import mm

print(f" Environment ready. morph {getattr(morph, '__version__', 'local_file')}")
```

```
 Environment ready. morph 1.1.9
```

6.3 Image Databases for Experimentation

The examples presented in this part of the book use, whenever possible, scanned documents, answer sheets, barcodes, *QRCodes*, and other images from real-world applications. To make the experiments fully reproducible — including in environments without internet access or access to the original MCTest files — we also employ public images widely used in teaching and research in Digital Image Processing and Computer Vision (DIP-CV).

Why Use a Benchmark Image Database?

Images such as `camera()` and `coins()` have been used for decades in textbooks, scientific articles, and teaching materials in DIP-CV. Their use offers important advantages:

- **Reproducibility:** any reader obtains exactly the same images, regardless of the computer or operating system used, without requiring external downloads or files specific to this book;
- **Comparability:** the results produced can be compared directly with those reported in the literature, since the same image sets are widely adopted as a reference;
- **Focus on algorithms:** because these images are compact, well-documented, and distributed without usage restrictions for educational and scientific purposes, they allow attention to be concentrated on processing techniques, reducing interference related to data acquisition or man-

agement.

6.3.1 Public Images with `skimage.data`

The `skimage.data` module provides a collection of reference images widely used in teaching, research, and algorithm validation activities in DIP-CV. Inspection of the `data.__all__` attribute shows that the current version contains **42 items**, including natural photographs, scanned documents, medical images, microscopy, textures, synthetic patterns, three-dimensional models, and temporal sequences. It should be noted that some of these items correspond to utility functions, such as `data_dir()` and `download_all()`, and not to images proper.

Since the purpose of this chapter is to present applications of document analysis and visual inspection, Table 6.2 gathers a **representative selection** of the most relevant images, organized according to their main applications in Computer Vision. Figure 6.1 presents a sample of these images, grouped according to the same classification adopted in the table.

Images used in this chapter

Although `skimage.data` provides dozens of reference images, only four are used directly in the experiments of this chapter. They were selected because they reproduce, in a controlled manner, characteristics frequently found in scanned documents and in industrial visual inspection systems. Table 6.1 summarizes the role of each of them throughout this chapter.

Table 6.1: skimage.data images used in the experiments of this chapter.

Image	Application in the chapter
`data.page()`	Scanned page used in the experiments on illumination correction, local enhancement (CLAHE), and automatic Otsu thresholding.
`data.text()`	Document containing printed text, used to illustrate segmentation, contour extraction, and typical OCR and OMR steps.
`data.coffee()`	Color photograph with natural illumination variations, used to exemplify techniques applicable to real, non-documentary scenes.
`data.brick()`	Reference texture used in examples of surface inspection and defect detection by local variance analysis.

Table 6.2: Selection of representative public images available in the *skimage.data* module, organized by application area.

Categoria	Função	Descrição
📄 Documentos & OCR/OMR	`data.page()`	Página digitalizada de documento — normalização de fundo, CLAHE e Otsu.
📄 Documentos & OCR/OMR	`data.text()`	Texto impresso — segmentação e extração de contornos em cenários de OCR/OMR.
🔵 Segmentação, Morfologia & Contornos	`data.coins()`	Conjunto de moedas — referência clássica para segmentação e *watershed*.
🔵 Segmentação, Morfologia & Contornos	`data.clock()`	Relógio analógico — detecção de formas e contornos.
🔵 Segmentação, Morfologia & Contornos	`data.binary_blobs()`	Blobs binários sintéticos — conectividade e morfologia matemática.

Categoria	Função	Descrição
🔵 Segmentação, Morfologia & Contornos	`data.moon()`	Superfície lunar — segmentação de crateras por relevo de intensidade.
Textura & Inspeção Industrial	`data.brick()`	Textura uniforme de tijolos — detecção de defeitos por variância local.
Textura & Inspeção Industrial	`data.checkerboard()`	Padrão xadrez — calibração de câmera e transformações geométricas.
🖼️ Fotografias Clássicas de PDI/VC	`data.camera()`	Fotógrafo com tripé — imagem de referência mais citada na literatura de PDI.
🖼️ Fotografias Clássicas de PDI/VC	`data.astronaut()`	Retrato colorido de astronauta — filtragem e realce em cor.
🖼️ Fotografias Clássicas de PDI/VC	`data.coffee()`	Xícara de café — cena real com variação de iluminação e cor.
🖼️ Fotografias Clássicas de PDI/VC	`data.cat()` / `data.chelsea()`	Fotografias coloridas de gatos — detecção de bordas e realce.
🖼️ Fotografias Clássicas de PDI/VC	`data.horse()`	Silhueta binária de cavalo — descritores de forma e contorno.

```python
import matplotlib.pyplot as plt
from skimage import data

# Images ordered by category; the title color reproduces the category color in the previous
tab.
imgs = [
    ("page",         data.page(),         "#2563eb"),   # Documents & OCR/OMR
    ("text",         data.text(),         "#2563eb"),
    ("coins",        data.coins(),        "#16a34a"),   # Segmentation & morphology
    ("binary_blobs", data.binary_blobs(), "#16a34a"),
    ("brick",        data.brick(),        "#ea580c"),   # Texture & industrial inspection
    ("checkerboard", data.checkerboard(),"#ea580c"),
    ("camera",       data.camera(),       "#7c3aed"),   # Classic computer vision/image
processing photographs
    ("coffee",       data.coffee(),       "#7c3aed"),
]

fig, ax = plt.subplots(2, 4, figsize=(11, 5.5))
for a, (nome, img, cor) in zip(ax.ravel(), imgs):
    a.imshow(img, cmap="gray")
    a.set_title(nome, color=cor, fontweight="bold")
    a.axis("off")
plt.tight_layout()
```

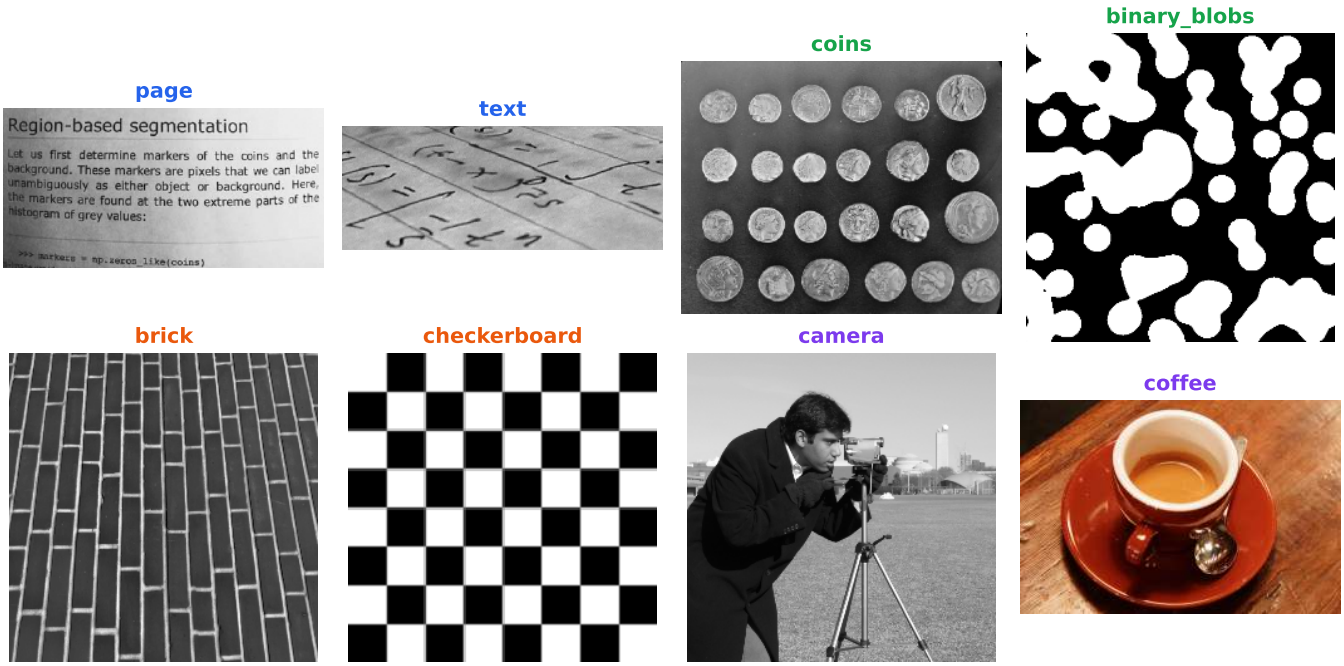


Figure 6.1: Sample of public images from *skimage.data*, grouped by application area.

6.4 Background Normalization and Local Contrast Equalization

Segmentation quality depends directly on the characteristics of the input image. In scanned documents, illumination variations, shadows, overexposed regions, and differences in paper tone reduce the contrast between the foreground and the background, making it difficult to apply global thresholding methods, such as the Otsu algorithm.

To minimize these effects, two complementary preprocessing techniques are employed:

- **Background normalization:** estimates the low-frequency component of the image through strong smoothing and then normalizes the original image with respect to this estimated background. This procedure reduces illumination gradients and compensates for slow intensity variations while preserving the structures of interest.
- **CLAHE** (*Contrast Limited Adaptive Histogram Equalization*), presented in **Chapter 4:** divides the image into small regions (*tiles*) and performs histogram equalization for each region independently. Contrast is limited to avoid excessive noise amplification, making the technique especially suitable for images with local illumination variations.

Figure 6.2 compares these strategies using the `page()` image from the `skimage.data` library. Six results are presented: (a) the original image; (b) direct binarization using the Otsu method, used as a reference; (c) the background estimated by Gaussian filtering; (d) the image after background normalization; (e) the binarization obtained after applying CLAHE followed by the Otsu method; and (f) the binarization obtained after background normalization followed by the application of the Otsu method.

The comparison allows one to observe the effect produced by each preprocessing stage and its influence on segmentation quality. In particular, background normalization reduces global illumination variations, while CLAHE increases local contrast between characters and background. Depending on the image characteristics, one or the other strategy may yield superior results, and there is no universally more suitable technique.

```
import cv2
from skimage import data
from morph import mm

img = data.page()  # or: img = mm.gray(img_final) - with exam sheet image
```

```
#   Method 1: Background normalization + Otsu
# Estimates the background with a large-sigma Gaussian filter (slow light variations)
# and divides pixel by pixel to cancel the illumination gradient
bg = cv2.GaussianBlur(img, (0, 0), sigmaX=25)
img_norm = cv2.divide(img, bg, scale=255)
img_norm_otsu = mm.threshold(img_norm)

#   Method 2: CLAHE + Otsu
# tileGridSize defines the size of each local region (tile)
# clipLimit controls the amplification ceiling - high values increase contrast
# but also amplify noise
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
img_clahe = clahe.apply(img)
img_clahe_otsu = mm.threshold(img_clahe)

#   Method 0: Direct Otsu (no preprocessing) - reference
img_otsu = mm.threshold(img)

mm.show(
    [img,  img_otsu, bg, img_norm, img_clahe_otsu, img_norm_otsu],
    titles=["(a) Original", "(b) Direct Otsu", "(c) Gaussian",  \
            "(d) Normalized background (img/bg)", "(e) CLAHE + Otsu", \
            "(f) Normalized background + Otsu"],
    cols=3,
    figsize=(14, 8)
)
```

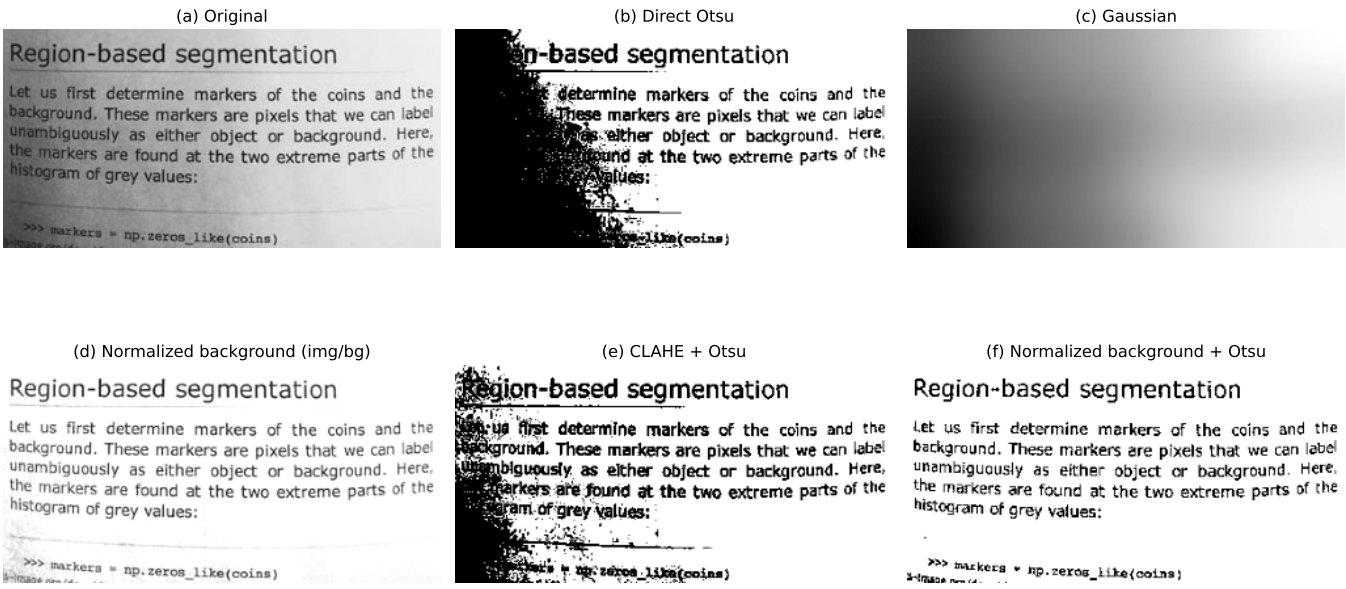


Figure 6.2: Comparison between preprocessing strategies for text image binarization: (a) original image; (b) direct thresholding by Otsu's method; (c) background estimated by Gaussian filtering; (d) image after background normalization (division by the smoothed image); (e) CLAHE followed by Otsu thresholding; and (f) background normalization followed by Otsu thresholding. The images illustrate the effect of each technique on compensating for illumination variations and on segmentation quality.

Why It Works — Background Normalization vs. CLAHE

Background normalization: by dividing the image by a heavily smoothed version of itself, slow variations in illumination (light gradient, edge shadow) are eliminated without affecting fine details —

text, lines, bubbles. The result is an image with approximately uniform lighting, where Otsu's global threshold works well across the entire page.
CLAHE: a globally equalized histogram "stretches" the tones of the entire image at once — useful when lighting is uniform, but problematic when it is not. CLAHE divides the image into small blocks (*tiles*) and equalizes each one separately, with a maximum amplification limit (`clipLimit`) to avoid amplifying noise. It is especially effective for enhancing locally underexposed regions, but it does not eliminate global gradients — thus, applying it after background normalization tends to produce more consistent results.

6.5 Optical Character Recognition (OCR)

After binarization, the next stage in document processing consists of converting the visual representation of characters into digitally encoded text, a process known as **Optical Character Recognition** (**OCR**).

In general terms, an OCR system comprises three stages:

1. **Segmentation:** identifies lines, words, and characters in the image, using horizontal and vertical projections or connected-component detection.
2. **Feature extraction:** represents each character through visual attributes, such as edges, curvatures, and stroke patterns.
3. **Recognition:** associates the extracted attributes with the most likely character. Current systems predominantly use recurrent neural networks (LSTM) or architectures based on *transformers*.

In this book, we use **Tesseract OCR**, accessed through the `pytesseract` library (installation: `pip install pytesseract`). Originally developed by Hewlett-Packard between 1985 and 1995 and currently maintained by Google, Tesseract is described in Smith (2007) and Smith (2013). In recent versions, text recognition is performed by LSTM neural networks.

OCR performance depends on the quality of the input image. Noise, low contrast, geometric distortions, and uneven illumination reduce the recognition rate. For this reason, steps such as deskewing, background normalization, and adaptive equalization (CLAHE) are part of image preprocessing.

Does Tesseract require a binarized image?

Tesseract internally incorporates an adaptive binarization step before character recognition. For this reason, providing the OCR with a previously binarized image does not always yield the best results. Since thresholding is an irreversible operation, it may eliminate subtle intensity variations at character edges, such as *antialiasing*, which can assist the recognition mechanism. In many cases, a grayscale image with good illumination and contrast produces a more faithful transcription than its binarized version.

To investigate this effect, we compare the text extracted by Tesseract from four versions of the same image, presented in Figure 6.3: (a) original image; (b) image subjected to adaptive equalization (CLAHE) followed by thresholding using Otsu's method; (c) image subjected to background normalization followed by thresholding using Otsu's method; and (d) image subjected only to background normalization, preserving grayscale tones.

The comparison between versions (c) and (d) shows that, in passages containing visually similar characters, the grayscale version (d) produced a transcription more faithful to the original text than the binarized version (c). This result indicates that thresholding applied during preprocessing can eliminate information useful for recognition. Thus, although binarization is essential for several image processing operations, it does not necessarily constitute the best input for OCR. The choice of preprocessing technique should consider the subsequent stage of the document pipeline.

```
import pytesseract
import shutil as _sh
```

```python
if _sh.which("tesseract") is None:
    # Build environment without Tesseract installed (e.g., without apt/sudo):
    # degrades instead of breaking the render. On Colab/local with Tesseract,
    # nothing changes.
    _AVISO_OCR = "[Tesseract OCR indisponivel neste ambiente - texto omitido]"
    pytesseract.image_to_string = lambda *a, **k: _AVISO_OCR
from skimage import data
import cv2
from morph import mm

img = data.page()

#   Reusing the results from the previous section
img_otsu = mm.threshold(img)

bg = cv2.GaussianBlur(img, (0, 0), sigmaX=25)
img_norm = cv2.divide(img, bg, scale=255)       # grayscale, without Otsu
img_norm_otsu = mm.threshold(img_norm)          # binarized

clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
img_clahe = clahe.apply(img)
img_clahe_otsu = mm.threshold(img_clahe)

#   Tesseract configuration
# --psm 6: assumes a single uniform block of text (suitable for the `page` image)
config = "--psm 6"

texto_original   = pytesseract.image_to_string(img, config=config)
texto_clahe_otsu = pytesseract.image_to_string(img_clahe_otsu, config=config)
texto_norm_otsu  = pytesseract.image_to_string(img_norm_otsu, config=config)
texto_norm_gray  = pytesseract.image_to_string(img_norm, config=config)

for nome, texto in zip(
    ["(a) Original", "(b) CLAHE + Otsu", "(c) Normaliz. fundo + Otsu",
     "(d) Normaliz. fundo (tons de cinza)"],
    [texto_original, texto_clahe_otsu, texto_norm_otsu, texto_norm_gray]
):
    print(f"--- {nome} ---")
    print(texto.strip(), "\n")

mm.show(
    [img, img_clahe_otsu, img_norm_otsu, img_norm],
    titles=["(a) Original", "(b) CLAHE + Otsu", "(c) Background norm. + Otsu",
            "(d) Background norm. (grayscale)"],
    cols=4,
    figsize=(16, 4)
)
```

```
--- (a) Original ---
[Tesseract OCR indisponivel neste ambiente - texto omitido]

--- (b) CLAHE + Otsu ---
[Tesseract OCR indisponivel neste ambiente - texto omitido]

--- (c) Normaliz. fundo + Otsu ---
[Tesseract OCR indisponivel neste ambiente - texto omitido]

--- (d) Normaliz. fundo (tons de cinza) ---
[Tesseract OCR indisponivel neste ambiente - texto omitido]
```

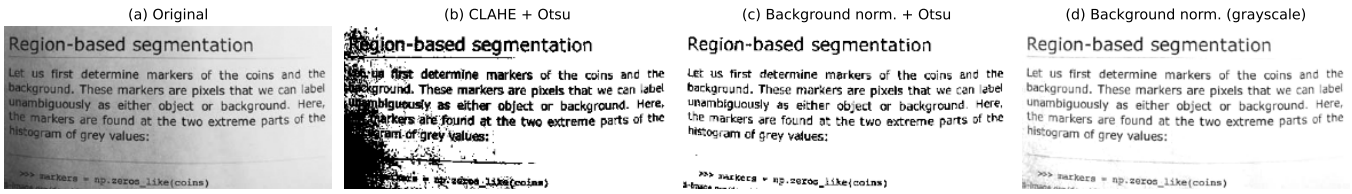


Figure 6.3: Comparison of the text extracted by Tesseract from four versions of the same image: (a) original image; (b) CLAHE followed by Otsu thresholding; (c) background normalization followed by Otsu thresholding; and (d) background normalization in grayscale, without thresholding. The comparison shows that external binarization does not always favor recognition, since Tesseract already performs its own adaptive binarization internally.

Why does preprocessing improve OCR?

OCR performance directly depends on the quality of the input image. Low contrast, uneven illumination, noise, and geometric distortions hinder the separation between text and background and increase the likelihood of recognition errors.
Techniques such as background normalization and adaptive equalization (CLAHE) correct illumination gradients and enhance local contrast between characters and background, producing images that are more suitable for automatic recognition. Thresholding, in turn, should be applied with caution: as it is an irreversible operation, it can eliminate subtle intensity variations at character edges—such as *antialiasing*—which the OCR engine itself uses internally to resolve ambiguities between visually similar symbols. For this reason, grayscale images, corrected only for illumination, often produce more faithful transcriptions than their binarized versions.
In documents captured by mobile device cameras, preprocessing tends to provide greater performance gains than in documents scanned with a *scanner*, where illumination is usually more uniform.

6.6 Automatic Translation of Recognized Text

After optical character recognition, the resulting text can be subjected to natural language processing techniques, such as spell checking, indexing, summarization, and machine translation.

Machine translation constitutes an independent stage of OCR. While OCR converts the characters present in the image into encoded text, translation operates on that text in the document's original language. Thus, recognition errors can be propagated to the translation, compromising the quality of the result. Current machine translation systems predominantly employ neural architectures based on attention mechanisms and *transformers* [Bahdanau; Cho; Bengio (2015); Vaswani (2017)].

In this example, the text obtained from the image subjected only to background normalization, without thresholding (item **d** of Figure 6.3), is used, as it presents the most faithful transcription among the strategies evaluated in the previous section.

The translation is carried out using the `deep-translator` library (installation: `pip install deep-translator`), which provides an interface to different machine translation services, including Google Translate.

```
from deep_translator import GoogleTranslator
import shutil as _sh

# Text obtained by OCR from the image with background normalization (grayscale)
texto_en = texto_norm_gray

if _sh.which("tesseract") is None:
    # Build environment without Tesseract installed (see previous cell): text_en
```

```
    # is already the OCR unavailable placeholder, not real text - skip translation
    # instead of failing when trying to translate a string that is not really English.
    texto_pt = "[Tradução indisponível neste ambiente - Tesseract OCR ausente]"
else:
    texto_pt = GoogleTranslator(source="en", target="pt").translate(texto_en)

print("--- Original text generated by the normalized grayscale image (OCR, EN) ---")
print(texto_en)

print("\n--- Translated text (PT-BR) ---")
print(texto_pt)

mm.show(
    [img_norm],
    titles=["Image with background normalization (grayscale)"],
    cols=1,
    figsize=(6, 4)
)
```

```
--- Original text generated by the normalized grayscale image (OCR, EN) ---
[Tesseract OCR indisponivel neste ambiente - texto omitido]

--- Translated text (PT-BR) ---
[Tradução indisponível neste ambiente - Tesseract OCR ausente]
```

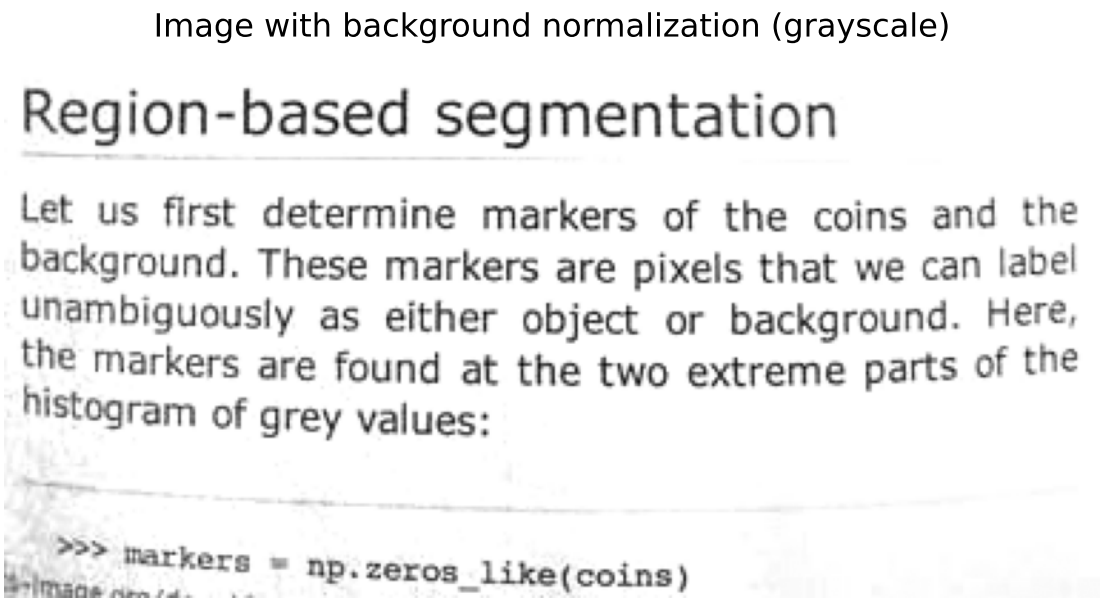


Figure 6.4: Automatic recognition and translation flow. The image pre-processed by background normalization, without thresholding, is used as input to Tesseract OCR, and the recognized text is translated from English to Portuguese using the *deep-translator* library.

Why does OCR quality influence translation?

Machine translation uses as input the text produced by OCR. Recognition errors, such as incorrect characters, incomplete, or fragmented words, are propagated to the translation stage and can alter the meaning of the text.
Consequently, the quality of the translation depends directly on the fidelity of the transcription obtained by OCR. As discussed previously, this fidelity is not always maximized by external binarization: grayscale images, corrected only for illumination, can preserve relevant information for distinguishing

visually similar characters. Thus, image preprocessing—and the appropriate choice of its steps based on the subsequent task—contributes to improving not only character recognition but also the performance of later natural language processing stages, such as translation, indexing, and summarization.

6.7 Fundamentals of OMR and Industrial Inspection

Optical Mark Recognition (OMR) is a Computer Vision technique aimed at the automatic identification of markings in predefined positions on a form. Its applications include answer sheets, questionnaires, administrative forms, and other structured documents.

Unlike OCR (*Optical Character Recognition*), which recognizes characters and words, OMR determines the presence, absence, or intensity of marks in previously known regions. Instead of interpreting text, it exploits geometric and statistical properties associated with the filling of these regions.

Modern OMR systems process images obtained from *scanners*, cameras, or mobile devices, automating tasks that previously relied on specialized equipment.

In general terms, an OMR system comprises the following stages:

1. **Acquisition:** conversion of the physical document into digital format;
2. **Pre-processing:** geometric correction, noise reduction, and binarization;
3. **Location of regions of interest:** identification of the areas intended for markings;
4. **Analysis of markings:** assessment of the filling of candidate regions;
5. **Interpretation:** conversion of markings into responses or structured data.

These principles extend naturally to **Automated Industrial Inspection**. On production lines, the same sequence — acquisition, pre-processing, segmentation, feature extraction, and decision — is employed to detect surface defects, verify component integrity, and measure dimensions with subpixel precision. The difference lies in the application domain: while OMR operates on documents with a predefined structure, industrial inspection deals with objects whose geometric and radiometric variations must be modeled more flexibly.

In the following sections, both applications are developed through practical projects that reproduce typical stages of real systems.

6.8 Practical Projects: Building a Document Analysis *Pipeline*

The concepts in this chapter will be developed through projects that reproduce typical stages of real document analysis systems, introducing techniques reusable in OCR, OMR, visual inspection, and form processing applications.

6.8.1 Automatic Document Alignment (*OCR/OMR Pre-processing*)

Deskew correction is a fundamental stage in document processing. Rotations introduced during scanning or capture compromise the location of regions of interest and reduce the accuracy of subsequent stages.

In this project, a system will be developed to automatically estimate the predominant orientation of the document and correct its skew. To this end, classical edge detection techniques using the Canny operator and line detection via the Hough Transform will be employed. From the identified lines, the rotation angle will be estimated, and an affine transformation will be applied to produce an aligned version of the document.

Since forms and answer sheets are frequently distributed in PDF format, the *pipeline* begins with the rasterization of each page, converting it into a matrix image. In this chapter, this stage will be performed using the `pdf2image` library, generating PNG images at 300 DPI (*dots per inch*) resolution. From these, the edge detection techniques, Hough Transform, segmentation, contour extraction, and automatic pattern recognition studied throughout the chapter can be applied.

```python
import os
import urllib.request
from pdf2image import convert_from_path
from skimage import data
import cv2

# Directory of microdata and answer sheets of the institutional exam
file_path = "dados/provas_qrcode_EP.pdf"
url_github = (
    "https://raw.githubusercontent.com/fzampirolli/"
    "pdi-vc/master/all/cap06/dados/provas_qrcode_EP.pdf"
)

# If the file does not exist locally, it is automatically downloaded from GitHub
if not os.path.exists(file_path):
    print(f"[DOWNLOAD] Downloading PDF from GitHub: {url_github}")
    try:
        # Ensures the 'data' folder exists before saving
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
        urllib.request.urlretrieve(url_github, file_path)
        print("[DOWNLOAD] PDF downloaded successfully!")
    except Exception as e:
        print(f"[DOWNLOAD] Failed to download the file: {e}")

print(f"PDF of scanned answer sheets: {file_path}")

if os.path.exists(file_path):
    # Rasterization of pages with optimized 300 DPI resolution
    pages = convert_from_path(file_path, dpi=300)
    for i, page in enumerate(pages):
        saida = f"test{i+1:02d}.png"
        page.save(saida)
        print(f"[INGESTION] PDF page converted successfully: {saida}")
else:
    print("[WARNING] PDF file not found in path. Enabling fallback via skimage.data.")
    # Injects public text matrix to ensure continuous pipeline execution
    img_fallback = data.text()
    cv2.imwrite("test02.png", img_fallback)
    print("[INGESTION] Structured fallback image: test02.png")

# Loads and displays the initial rasterized image using the morph ecosystem
if os.path.exists('test02.png'):
    img_original = mm.read('test02.png')
else:
    # Definitive fallback in case even skimage fails
    img_original = np.ones((400, 400), dtype=np.uint8) * 255

mm.show(img_original, figsize=(4, 3))
```

```
PDF of scanned answer sheets: dados/provas_qrcode_EP.pdf
[INGESTION] PDF page converted successfully: test01.png
[INGESTION] PDF page converted successfully: test02.png
[INGESTION] PDF page converted successfully: test03.png
```

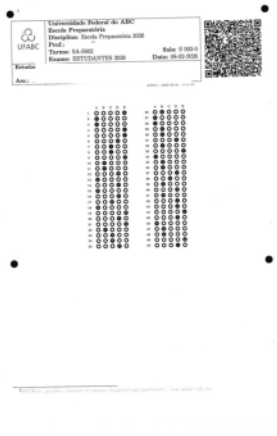


Figure 6.5: *Pipeline* for document ingestion: adaptive rasterization of PDF pages into discrete arrays in PNG format, showing page two.

6.8.2 Deskew Algorithm

The deskew stage aims to estimate and correct the global tilt of a scanned document, aligning its content with the image axes. Figure 6.7 presents the complete processing flow, from the original image to the result after geometric correction. Additionally, the simulator in Figure 6.6 allows visualizing how the Hough Transform works and understanding how the predominant orientation is estimated.

The procedure consists of three main stages:

1. edge detection using the Canny operator;
2. estimation of the predominant orientation through the Linear Hough Transform;
3. tilt correction using an affine rotation transformation.

After rectification, the document assumes an approximately horizontal orientation, favoring the subsequent stages of segmentation, connected component labeling, and recognition of characters and marks.

6.8.3 Mathematical Modeling

The following subsections formalize, in mathematical terms, the steps described previously, relating the image gradient, the parametrization of lines in the Hough space, and the rotation matrix used in the geometric correction.

6.8.3.1 Edge Detection

Initially, the image is smoothed by a Gaussian filter, reducing the effect of high-frequency noise that can generate spurious edges. The concepts of spatial filtering and convolution were presented in **Chapter 3**.

Next, the Canny operator estimates the image gradient. Let $f(x, y)$ be the image intensity and α the rotation angle.

The gradient magnitude is given by

$$|\nabla f(x, y)| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}.$$

where:

- $f(x, y)$ represents the image intensity at position (x, y);
- $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ are the partial derivatives in the horizontal and vertical directions;

- $|\nabla f(x, y)|$ is the gradient magnitude.

After computing the gradient, the algorithm applies non-maximum suppression and hysteresis thresholding, producing a binary image containing the main edges of the document.

6.8.3.2 Hough Transform

The binary edge image is processed by the Linear Hough Transform, whose objective is to detect approximately straight structures. Instead of the Cartesian representation of the line, $y = ax + b$, the normal form representation is used,

$$\rho = x \cos\theta + y \sin\theta,$$

where:

- x and y are the coordinates of a point belonging to the line;
- ρ is the perpendicular distance between the line and the origin of the image coordinate system;
- θ is the angle formed between the normal to the line and the horizontal axis of the image.

In this representation, each edge point (x, y) generates a curve in the parameter space (ρ, θ). The intersection of the curves produced by points belonging to the same line gives rise to maxima in a two-dimensional array called the **accumulator**. Thus, the peaks of the accumulator correspond to the predominant lines in the image, such as document edges, form lines, or text lines.

To estimate the global skew of the document, only the lines whose angles satisfy

$$-45^\circ \leq \theta \leq 45^\circ$$

are considered. This restriction eliminates orientations incompatible with the expected layout of the document and reduces the influence of vertical lines or irrelevant structures. Let $\theta_1, \theta_2, \ldots, \theta_n$ be the set of angles of the selected lines. The global skew estimate is obtained by the median,

$$\hat{\theta} = \operatorname{med}(\theta_1, \theta_2, \ldots, \theta_n),$$

where:

- θ_i is the angle of the i-th line detected by the Hough Transform;
- n is the number of lines considered after angular filtering;
- $\hat{\theta}$ is the estimate of the global skew of the document.

The median is adopted because it is less sensitive to the presence of isolated detections (*outliers*) than the arithmetic mean, yielding a more stable estimate of the predominant orientation.

6.8.3.3 Affine Rotation

Let $\hat{\theta}$ be the slope estimated in the previous step. The geometric correction consists of applying an affine rotation transformation around the center of the image, so that the predominant orientation coincides with the horizontal axis. Affine transformations were studied in **Chapter 2**, along with translation, scaling, shearing, and rotation operations.

Let (x, y) be the position of a pixel relative to the center of the image and (x', y') its position after rotation. The transformation is described by

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = R(\alpha) \begin{bmatrix} x \\ y \end{bmatrix},$$

where

$$R(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix},$$

with:

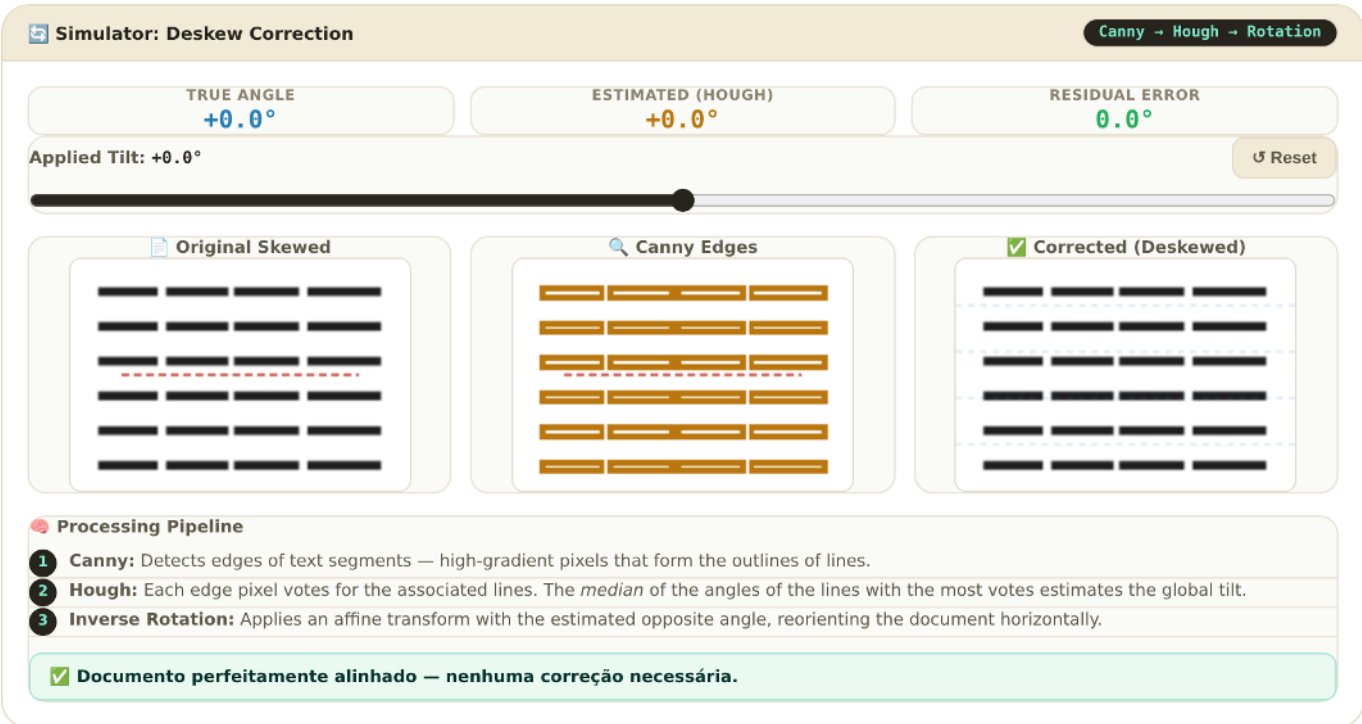
- (x, y) being the original pixel coordinates relative to the center of the image;
- (x', y') being the pixel coordinates after rotation;
- α being the rotation angle applied to compensate for the estimated document slope;
- $R(\alpha)$ being the rotation matrix.

In practice, the applied angle corresponds to the opposite of the estimated slope,

$$\alpha = -\hat{\theta},$$

where $\hat{\theta}$ represents the predominant orientation obtained by the Hough Transform.

Since the transformed coordinates do not always coincide with integer positions in the pixel grid, it is necessary to resample the image to determine the new intensity values. In the implementation presented in this chapter, the `mm.rotate` function performs this operation using bicubic interpolation, reducing resampling artifacts and preserving the visual continuity of edges and characters.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap06/sim-06-deskew.html

Figure 6.6: Interactive skew correction simulator (*deskew*): move the *slider* to tilt the document and observe the three steps of the *pipeline* — tilted image, Canny edges, and corrected result.

```
def retificar_inclinacao_documento(img):

    gray = mm.gray(img) if img.ndim == 3 else img
    edges = cv2.Canny(cv2.GaussianBlur(gray, (5,5), 0), 50, 150)
    lines = cv2.HoughLines(edges, 1, np.pi/180, 200)

    if lines is None:
        return edges, img

    angulos = []
```

```
    for line in lines:
        angulo = np.rad2deg(line[0][1]) - 90
        if -45 < angulo < 45:
            angulos.append(angulo)

    if not angulos:
        return edges, img

    return edges, mm.rotate(img, np.median(angulos), interp="bicubic")

# Running the deskew pipeline
img_edges, img_final = retificar_inclinacao_documento(img_original)

# Standardized multiple display using the book's native format
mm.show(
    [img_original, img_edges, img_final],
    titles=["Original Image", "Canny Edges", "Rectified Document"],
    cols=3,
    figsize=(12, 4)
)
```

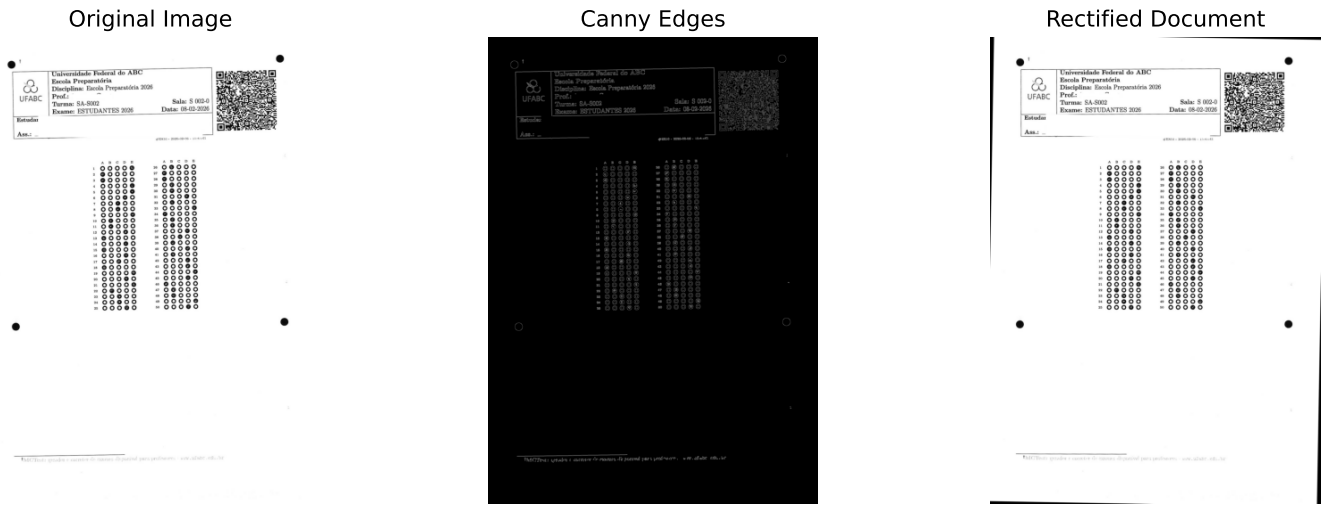


Figure 6.7: *Axial rectification pipeline*: comparative display between the original rotated input, the Canny structural gradient map, and the final aligned result with normalized white background.

> ℹ 🧠 Why It Works — Hough Transform
>
> In the Hough Transform, each edge pixel contributes votes to all lines that can pass through its position. Instead of selecting only the line with the highest number of votes, the algorithm considers all lines whose vote count exceeds a minimum threshold and calculates their respective angles. The global document skew is then estimated by the median of these angles, a measure that is robust to outliers. Thus, spurious lines produced by shadows, noise, or other elements of the image exert little influence on the final estimate, as long as the majority of detected lines corresponds to the document edges.

6.8.4 Practical Limitations

Although it performs well under typical scanning conditions, the method relies on the existence of sufficiently well-defined linear structures that can be detected by the Hough Transform, such as page edges, form lines, or text lines. Its accuracy may be reduced in images with low resolution, excessive noise, strong shadows, or large skew angles. In general, scanned documents at resolutions near 300 DPI with uniform lighting yield adequate results for OCR and OMR applications.

The implementation presented in this chapter is intended for didactic purposes, illustrating the principles of automatic skew correction through edge detection, the Hough Transform, and affine rotation. Because it uses only the orientation of predominant linear structures, the method can be applied to different types of documents without relying on specific markers.

In real document analysis systems, however, alignment typically uses previously known geometric markers. In the answer sheet model employed by the MCTest ecosystem, for example, four black reference discs are used, in addition to the regions corresponding to the header, the *QRCode*, and the answer boxes. Locating these elements allows the rotation, scale, and translation of the sheet to be estimated simultaneously, making registration less sensitive to the amount of text, the absence of structural lines, and variations in printing or scanning.

For this reason, the Hough Transform-based approach is used in this chapter to introduce the fundamentals of the problem, while subsequent stages adopt alignment by geometric markers, the predominant strategy in OMR and document analysis systems.

6.8.5 Edge and Contour Detection

The precise localization of regions of interest is an essential step in OMR systems. In the answer sheet model used in this chapter, the header and the answer grid are contained within a virtual rectangle delimited by four black discs positioned at the corners. The identification of these markers allows the region of interest to be located and geometric distortions introduced during image acquisition to be corrected.

The procedure comprises five steps. Initially, a **morphological closing** (dilation followed by erosion) is applied, an operation studied in **Chapter 4**, using a disk-shaped structuring element (`mm.sedisk(33)`). This operation reduces small discontinuities and preserves the reference discs, making them more homogeneous. Next, the image is inverted (`mm.neg`), so that the discs become light components against a dark background.

In the following step, the operation `mm.edgeoff` (**Chapter 4**) is applied, which removes components connected to the image borders, eliminating artifacts such as scanning shadows, crop marks, and other spurious objects at the margins. The remaining components are then analyzed based on their contours and filtered by geometric properties, such as area and circularity, to identify candidate discs. The MCTest implementation makes this process more robust by selecting, among all candidates, the four whose centers form a rectangle with width compatible with that of the image, reducing the occurrence of false positives.

Finally, the centers of the four discs are spatially ordered (top left, top right, bottom left, and bottom right) and used as control points in a perspective transformation (*perspective warp*). This transformation rectifies the image, producing an aligned representation with known dimensions, suitable for the subsequent segmentation and recognition steps.

The main steps of this *pipeline*, from morphological processing to the rectified image, are illustrated below.

```
import cv2
import numpy as np
from morph import mm

# img: grayscale image of the answer sheet
if img_final.ndim == 2:
    img = img_final
else:
    img = mm.gray(img_final)

# 1. Morphological closing: preserves dark discs, removing everything smaller than the disc
img_close = mm.close(img, mm.sedisk(41))

# 2. Inversion: dark discs become bright components on a dark background
img_neg = mm.neg(img_close)

# 3. Removes connected components touching the image border
```

```
img_edgeoff = mm.edgeoff(img_neg)

# 4. Extraction of external contours
contornos, _ = cv2.findContours(img_edgeoff, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# 5. Filtering by area and circularity, keeping only the 4 discs
centros = []
for c in contornos:
    area = cv2.contourArea(c)
    perimetro = cv2.arcLength(c, True)
    if area < 50 or perimetro == 0:
        continue
    circularidade = 4 * np.pi * area / (perimetro ** 2)
    if circularidade > 0.8:
        M = cv2.moments(c)
        cx, cy = M["m10"] / M["m00"], M["m01"] / M["m00"]
        centros.append((cx, cy))

# Robust verification: interrupts the pipeline with a clear message instead of AssertionError
if len(centros) != 4:
    print(f"[WARNING] Expected 4 marker discs, found {len(centros)}.")
    print("  Check whether the image is a valid MCTest answer sheet")
    print("  or adjust the circularity and minimum area parameters.")
    img_retificada = img  # fallback: preserves the image without rectification
else:
    # 6. Sorting of centers: top-left, top-right,
    # bottom-left, bottom-right
    pts = np.array(centros, dtype=np.float32)
    soma = pts.sum(axis=1)
    diff = pts[:, 0] - pts[:, 1]
    tl = pts[np.argmin(soma)]
    br = pts[np.argmax(soma)]
    tr = pts[np.argmax(diff)]
    bl = pts[np.argmin(diff)]
    pts_ordenados = np.array([tl, tr, bl, br], dtype=np.float32)

    # 7. Rectification by perspective transformation (warp)
    largura, altura = 800, 800
    destino = np.array(
        [[0, 0], [largura, 0], [0, altura], [largura, altura]], dtype=np.float32
    )
    M_persp = cv2.getPerspectiveTransform(pts_ordenados, destino)
    img_retificada = cv2.warpPerspective(img, M_persp, (largura, altura))

    mm.show(
        [img_close, img_edgeoff, img_retificada],
        titles=["Closing (sedisk 41)", "edgeoff", "Rectified (warp)"],
        cols=3,
        figsize=(12, 4)
    )

mm.write(img_retificada, "img_beetween_disks.png")
```

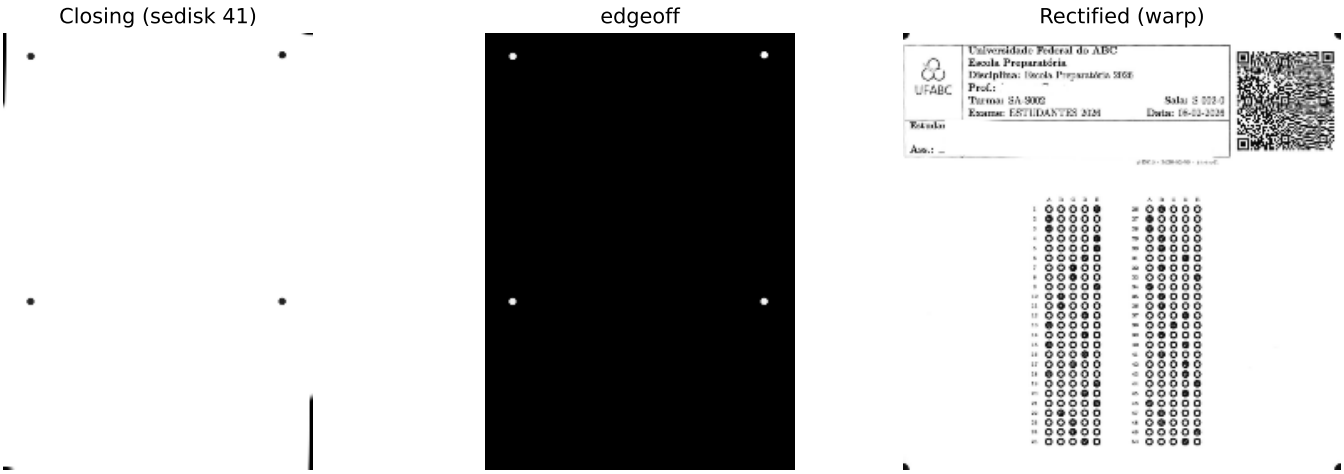


Figure 6.8: Detection of marker discs, contour extraction, and rectification by perspective transformation.

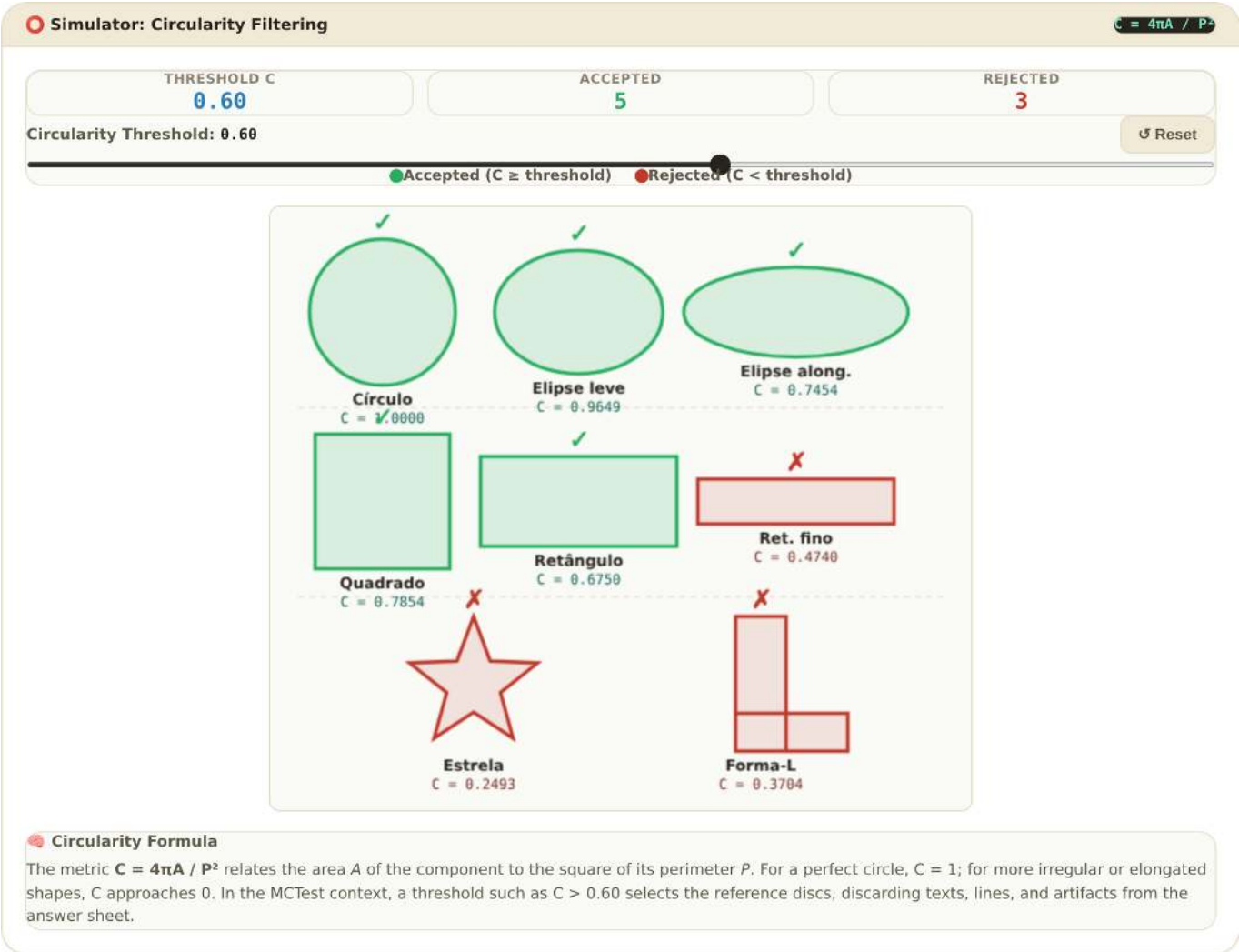
> **Why It Works — From Morphological Closing to Rectification**
>
> **Morphological closing:** dilation followed by erosion fills small discontinuities and smooths the contours of objects without significantly altering their overall shape. By using a large structuring element (`sedisk(41)`), fine details such as texts, form lines, and small noise tend to be incorporated into the background during processing, while larger-scale objects, such as the reference disks, remain preserved and become more homogeneous.
> **Circularity:** after isolating the candidate components, the metric $C = \frac{4\pi A}{P^2}$ quantifies how closely their shape approximates a circle. Its value is equal to 1 for a perfect circle and decreases as the contour becomes more irregular. Thus, a threshold such as $C > 0{,}8$ allows discarding most false positives without resorting to learning models. A simulator of this metric is presented in Figure 6.9.
> **Perspective warp:** once the four reference disks have been identified, their centers are used as control points to estimate the projective transformation that maps the captured image onto the document plane. This transformation corrects the distortions introduced by perspective during image acquisition, producing a frontal representation with known dimensions and suitable for the subsequent segmentation and recognition stages.

6.8.6 Isolation, Segmentation, and Decoding of the *QRCode*

After the geometric rectification of the answer sheet, detection and decoding of the *QRCode* present in the form are performed. This marker stores information used by the OMR (*Optical Mark Recognition*) system, such as the student's identification, the exam code, and its variation, enabling the retrieval of the corresponding answer key from the database. For security reasons, this information is encrypted before generating the *QRCode*. Thus, the decoded sequence corresponds to a hexadecimal *string*, whose interpretation is performed exclusively by the MCTest system. The procedure consists of three stages: morphological preprocessing, isolation of the *QRCode* region, and decoding of its content.

Initially, the rectified grayscale image is binarized using the `mm.threshold` operation. Next, a **morphological opening** (erosion followed by dilation), studied in **Chapter 4**, is applied using a square structuring element (`mm.sebox(2)`). This operation removes small noises and smooths imperfections without compromising the marker's structure. Finally, the image is inverted (`mm.neg`), so that the *QRCode* becomes a light component against a dark background, facilitating the extraction of its contours.

The location of the *QRCode* is carried out by analyzing the external contours of the binarized image. Among the detected components, the one with the largest area and approximately square geometry is selected, discarding the other printed elements on the sheet. Subsequently, the corresponding region is expanded by a small safety margin, ensuring the complete preservation of the marker.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap06/sim-06-circularidade.html

Figure 6.9: Interactive simulator of circularity filtering: move the *slider* to adjust the threshold C and observe which components are accepted (green) or rejected (red).

The *QRCode* is then extracted directly from the rectified grayscale image, preserving its radiometric quality. Since this region generally presents reduced dimensions, resizing with cubic interpolation is applied, increasing the spatial resolution and favoring the identification of its modules. The reading is performed by the OpenCV *QRCode* detector (`cv2.QRCodeDetector`), which retrieves the originally encoded character sequence.

The complete flow of this processing, from morphological preprocessing to *QRCode* decoding, is illustrated in Figure 6.10. The excerpt shown in the output corresponds only to the beginning of the encrypted hexadecimal *string*; its complete interpretation is performed internally by MCTest after decoding.

```
import cv2
import numpy as np
from morph import mm

# f: rectified image converted to the correct 8-bit type (0-255)
f = img_retificada.astype('uint8')

# 1. Thresholding: converting the grayscale image to binary
f_thresh = mm.threshold(f)

# 2. Morphological opening: removes small noise and smooths block contours
f_open = mm.open(f_thresh, mm.sebox(2))

# 3. Morphological inversion: dark modules become light components on a dark background
f_inv = mm.neg(f_open)

# 4. Safe conversion to uint8 with 0-255 scale
img_uint8 = (f_inv.astype(np.uint8) * 255) if f_inv.max() == 1 else f_inv.astype(np.uint8)

# 5. Detection of external contours
contornos, _ = cv2.findContours(img_uint8, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

if not contornos:
    raise ValueError("Nenhum contorno encontrado. Verifique o limiar ou a imagem de entrada.")

# 6. Filtering by the largest contour with an approximately square proportion
#    (aspect ratio between 0.7 and 1.3 discards elongated rectangles from the sheet)
def is_square_like(contorno, tol=0.3):
    x, y, w, h = cv2.boundingRect(contorno)
    ratio = w / h if h > 0 else 0
    return (1 - tol) <= ratio <= (1 + tol)

candidatos = [c for c in contornos if is_square_like(c)]

if not candidatos:
    raise ValueError(
        "Nenhum contorno quadrado encontrado. "
        "Verifique se o QRCode está presente na imagem ou ajuste a tolerância."
    )

# Selects the largest square candidate by bounding box area
maior_contorno = max(candidatos, key=lambda c: cv2.boundingRect(c)[2] *
cv2.boundingRect(c)[3])
x, y, w, h = cv2.boundingRect(maior_contorno)

# 7. Expansion of the bounding box with a safety margin (prevents QRCode truncation)
margem = 5
h_img, w_img = img_uint8.shape[:2]
x1 = max(x - margem, 0)
y1 = max(y - margem, 0)
x2 = min(x + w + margem, w_img)
y2 = min(y + h + margem, h_img)
```

```python
# 8. Cropping the region of interest from the original image (sharp, grayscale)
img_qrcode_final = img_retificada[y1:y2, x1:x2]

# 9. Enlargement to minimum decoding resolution (400 px on the longer side)
#    cv2.QRCodeDetector requires modules at least 3-4 px wide to decode
#    reliably; images smaller than ~400 px tend to fail.
lado = max(img_qrcode_final.shape[:2])
escala = max(400 / lado, 1.0)
img_para_leitura = cv2.resize(
    img_qrcode_final, None,
    fx=escala, fy=escala,
    interpolation=cv2.INTER_CUBIC
)

# 10. Initialization of OpenCV's native QRCode detector
detector = cv2.QRCodeDetector()

# 11. Geometric detection and decoding of the textual data
dados, pontos, qrcode_reto = detector.detectAndDecode(img_para_leitura)


# Intermediate visualization: progression from binarization to isolation of the QRCode
mm.show(
    [f_thresh, f_open, f_inv, img_qrcode_final],
    titles=["1. Thresholding", "2. Morphological opening", "3. Inversion", "Final image"],
    cols=3, figsize=(12, 4)
)

# Validation and output of extracted metadata
if dados:
    print(f"QRCode successfully decoded: \n{dados[:50]}...")
else:
    raise ValueError(
        "Falha na decodificação do QRCode. "
        "Verifique o limiar, as margens da região ou a qualidade da imagem."
    )
```

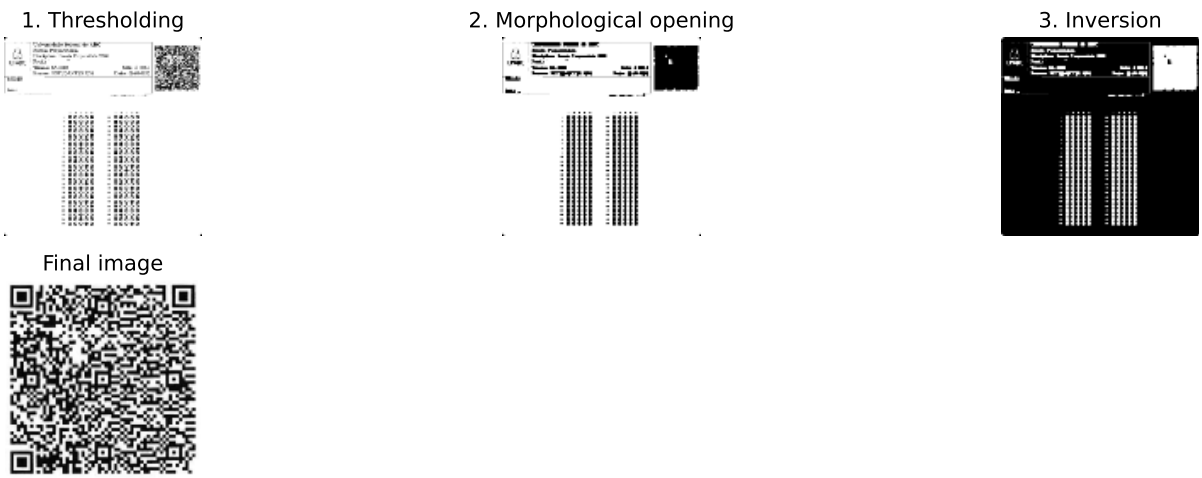
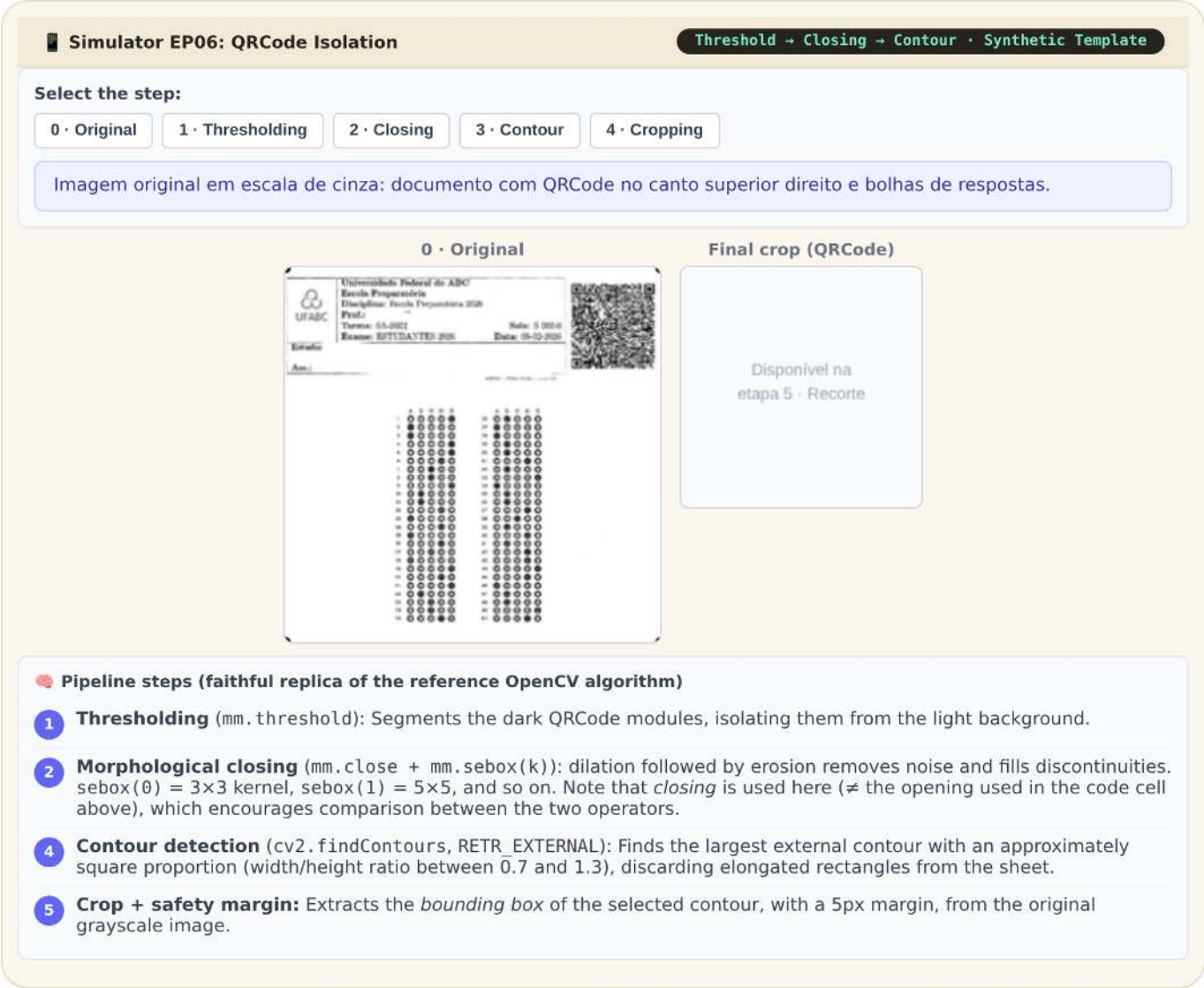


Figure 6.10: *Pipeline* of *QRCode* processing: thresholding, morphological opening, inversion, and final cropping for decoding.

```
QRCode successfully decoded:
325a356b71367266556955646b7233454149624a694f417730...
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap06/sim-06-qrcode.html

Figure 6.11: Interactive simulator of the QRCode isolation *pipeline*: navigate through the filtering steps, adjust the structuring element of the morphological closing, and see the square contour detection on the original template image.

Why It Works — Isolating and Decoding the *QRCode*

Morphological opening: unlike closing, opening (erosion followed by dilation) removes small noise and protrusions without significantly altering the geometry of larger objects. Thus, it preserves the structure of the *QRCode* while eliminating spurious components that could hinder its localization.
Geometry-based selection: the *QRCode* has an approximately square shape ($w/h \approx 1$). Combining this criterion with the selection of the largest-area component discards form lines, texts, and other printed elements, allowing the marker to be isolated without the use of learning models.
Resizing before decoding: when the *QRCode* occupies few pixels in the image, its modules become difficult to distinguish. Resizing with cubic interpolation increases the spatial resolution of the region of interest, facilitating the identification of the code patterns by `cv2.QRCodeDetector` and making decoding more robust.

6.8.7 Barcode Decoding

In the previous section, the decoding of *QRCodes* was performed using OpenCV's native detector (`cv2.QRCodeDetector`), which integrates into a single interface the geometric detection of the symbol, its rectification, and the extraction of the encoded information.

For linear (1D) barcodes, such as EAN-13, Code 39, and Code 128, a widely used alternative is the `pyzbar` library. Unlike `QRCodeDetector`, it supports various barcode symbologies and can also be employed for reading *QRCodes*.

The procedure consists of automatically locating each symbol present in the image and interpreting the corresponding sequence of bars and spaces, yielding the encoded character string. In addition to the decoded data, the library provides information such as the identified symbology and the code's position in the image, enabling its subsequent validation or processing.

Figure 6.12 presents an example of a barcode and the result of its decoding using the `pyzbar` library.

```
import os
import urllib.request
import numpy as np
from pyzbar.pyzbar import decode
from morph import mm

barcode_path = 'dados/barcode.png'
url_github = (
    "https://raw.githubusercontent.com/fzampirolli/"
    "pdi-vc/master/all/cap06/dados/barcode.png"
)

# If the file does not exist locally, downloads it automatically from GitHub
if not os.path.exists(barcode_path):
    print(f"[DOWNLOAD] Downloading barcode from GitHub: {url_github}")
    try:
        os.makedirs(os.path.dirname(barcode_path), exist_ok=True)
        urllib.request.urlretrieve(url_github, barcode_path)
        print("[DOWNLOAD] Image downloaded successfully!")
    except Exception as e:
        print(f"[DOWNLOAD] Failed to download file: {e}")

if os.path.exists(barcode_path):
    image = mm.read(barcode_path)
else:
    # Synthetic fallback: generates a vertical bar pattern that simulates a Code-128
    print("[WARNING] File 'dados/barcode.png' not found.")
    print("        Using synthetic image for pipeline demonstration.")
```

```
    h, w = 100, 400
    img_synth = np.ones((h, w), dtype=np.uint8) * 255
    # Dark bars at regular positions (simplified pattern)
    for x in range(20, w - 20, 8):
        if (x // 8) % 3 != 0:
            img_synth[:, x:x+4] = 0
    image = img_synth

mm.show(image)

# Runs pyzbar decode
try:
    barcodes = decode(image)
except ImportError:
    print("[ERROR] System zbar not found. Run: !apt-get install -y libzbar0")
    barcodes = []

if barcodes:
    dados_bc = barcodes[0].data.decode("utf-8")
    tipo = barcodes[0].type
    print(f"Barcode decoded successfully [{tipo}]:\n{dados_bc}")
else:
    print("[INFO] No barcode detected in the image.")
    print(
        "In a synthetic image this is expected - "
        "replace with the real file to decode."
    )
```

Figure 6.12: Decoding linear barcodes with *pyzbar*: input image and extracted data.

```
Barcode decoded successfully [EAN13]:
0000000000055
```

6.9 The MCTest as a Case Study: From Prototype to Production System

Up to this point, the main stages of the processing *pipeline* have been presented and analyzed individually, including skew correction (*deskew*), marker detection, perspective rectification, and *QRCode* reading. Although this approach facilitates the understanding of each technique, real-world applications require the integration of these stages into a single, consistent processing flow.

The **MCTest** constitutes an example of such integration. Developed at UFABC and made available as open-source software, the system has been used since 2012 for the automated grading of assessments, offering support for different answer sheet models, individualized answer keys, and automatic generation of performance reports (ZAMPIROLLI, 2023).

From this point onward, the focus shifts from the isolated implementation of algorithms to the organization of these algorithms within a complete application. In addition to the quality of image processing methods, a system of this nature must meet requirements such as robustness under different acquisition conditions, ease of maintenance, and the capacity to evolve toward new functionalities.

In the following sections, the Computer Vision module of MCTest, implemented in the file `CVMCTest.py`, will be examined. The goal is to show how the concepts presented throughout this chapter are combined into a processing *pipeline* employed in a real-world application.

6.9.1 Obtaining and Preparing the Module

The `CVMCTest.py` file is part of the MCTest system and, in its original version, depends on models, configurations, and other components of the Django *framework*. Since these dependencies are not available in the environment used in this chapter, the module needs to be adapted to run independently.

To this end, the file is obtained directly from the project repository using the `requests` library. Subsequently, `sed` commands are employed to remove imports and dependencies specific to the Web environment, producing a self-contained version of the module. This adaptation preserves the implementation of the Computer Vision algorithms, allowing their execution and analysis without the need to install or configure the entire MCTest system infrastructure.

```
import requests
CVMCTest = requests.get(
    "https://raw.githubusercontent.com/fzampirolli/mctest/master/exam/CVMCTest.py"
    )
with open('CVMCTest.py', 'w') as writefile:
    writefile.write(CVMCTest.text)
```

Each `sed` command removes specific imports from the Django environment, making the `CVMCTest.py` file independently usable in this chapter.

```
# remove lines with "form django.", ...
!sed --in-place '/from django./d' CVMCTest.py
!sed --in-place '/from exam./d' CVMCTest.py
!sed --in-place '/from mctest./d' CVMCTest.py
!sed --in-place '/from student./d' CVMCTest.py
!sed --in-place '/from topic./d' CVMCTest.py
!sed --in-place '/from .models import VariationExam/d' CVMCTest.py
```

Why Remove Django Dependencies?

In the original implementation, the `CVMCTest.py` file is part of an application developed with the Django *framework* and, therefore, imports models, settings, and other components specific to that environment. Since these elements are not available in this chapter, the module cannot be imported directly.
The `sed` commands remove only these dependencies, without altering the Computer Vision routines implemented in the file. This way, the module can be run independently, preserving the behavior of the presented algorithms.
This procedure illustrates an important software engineering principle: separating application logic from the infrastructure in which it is embedded, facilitating reuse, testing, and the study of specific components.

6.9.2 Extraction of the Answer Area

After reading the sheet, the `getAnswerArea` function automatically executes the steps for detecting reference markers and perspective correction presented in the previous sections. As a result, an image containing only the region intended for responses is obtained, aligned and with standardized dimensions.

This standardization simplifies the subsequent processing steps, as the location of the marking fields becomes known and independent of the original position of the sheet during scanning.

Figure 6.13 presents the original answer sheet in grayscale, while Figure 6.14 shows the answer region obtained after applying the `getAnswerArea` function.

```
import os
from pdf2image import convert_from_path
```

```python
from skimage import data as skdata
import cv2
from morph import mm

file = "dados/provas_qrcode_EP.pdf"
MYFILES = 'extra02.qrcode'

if os.path.exists(file):
    pages = convert_from_path(file, 200)  # dpi 100=min 500=max
    numPAGES = 0
    for page in pages:
        myfile0 = MYFILES + '_p' + str(numPAGES) + '.png'
        page.save(myfile0)
        numPAGES += 1
        print(f"[INGESTION] Page converted: {myfile0}")
    pages.clear()
    img_color = mm.read(myfile0)
    img_inicial = mm.gray(img_color)
else:
    print("[WARNING] File 'dados/provas_qrcode.pdf' not found.")
    print("        Using public image skimage.data.page() as substitute.")
    img_inicial = skdata.page()

mm.show(img_inicial)
```

```
[INGESTION] Page converted: extra02.qrcode_p0.png
[INGESTION] Page converted: extra02.qrcode_p1.png
[INGESTION] Page converted: extra02.qrcode_p2.png
```

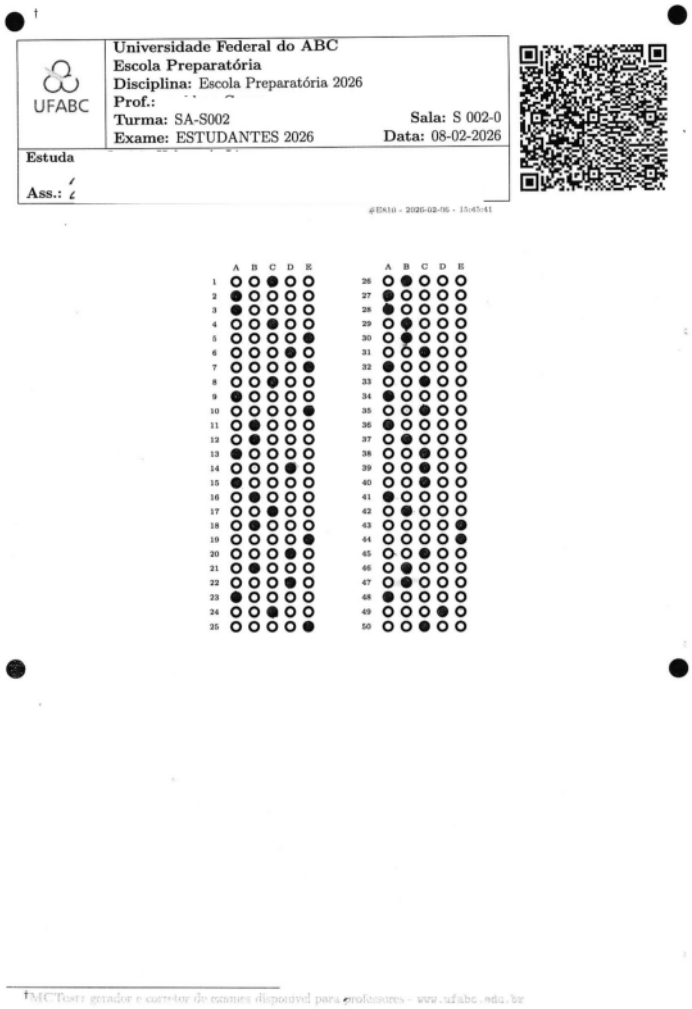


Figure 6.13: Image of the grayscale answer sheet loaded from the rasterized PDF.

```
import CVMCTest
countPage = 0
img_getAnswerArea = CVMCTest.cvMCTest.getAnswerArea(img_inicial, countPage)
mm.show(img_getAnswerArea)
```

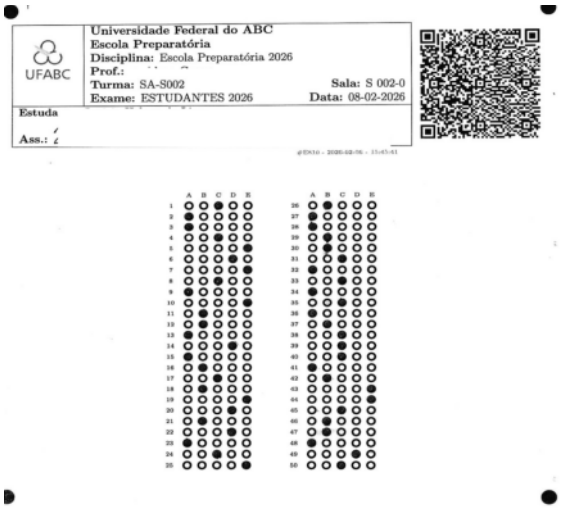


Figure 6.14: Answer area extracted by *getAnswerArea*: rectified region containing the marking frames.

Compatibility note: recent versions of NumPy (2.0) have removed the alias `np.int0`. If `CVMCTest.py` uses this type, the command below applies the correction directly to the file before reloading it:

```
!sed -i 's/box = np.int0(cv2.boxPoints(rect))/box = cv2.boxPoints(rect).astype(np.intp)/' \
    ./CVMCTest.py
```

```
import importlib
import CVMCTest

importlib.reload(CVMCTest)
```

```
<module 'CVMCTest' from '/home/fz/VSCode/pdi-vc/gen/quarto/py.en/cap06/CVMCTest.py'>
```

6.9.3 *QRCode* Segmentation

After extracting the answer area, MCTest performs two preparatory steps for reading the markings: *QRCode* segmentation and locating the frames that contain the questions.

The `segmentQRcode` function isolates the image region corresponding to the *QRCode*, shown in Figure 6.15. This region is then processed by the `getQRCode` function, which is responsible for decoding it and extracting the exam metadata.

If the *QRCode* cannot be decoded, the sheet processing continues normally. The student's answers are still read and recorded in the output CSV file; only the information obtained from the *QRCode*, such as the exam or student identification, remains unavailable.

```
import CVMCTest
imgQRcode = CVMCTest.cvMCTest.segmentQRcode(img_getAnswerArea, countPage)
mm.show(imgQRcode)
```

Figure 6.15: Region of the *QRCode* isolated by *segmentQRcode* within the rectified response area.

The function `CVMCTest.cvMCTest.getQRCode(img, countPage)` integrates the QRCode segmentation and decoding steps. Internally, it uses `CVMCTest.cvMCTest.decodeQRcode(imgQRcode)` to interpret the hexadecimal string encoded in the symbol and build the `qr` dictionary, in addition to returning the logical indicator `myFlagArea`, which reports whether the reading was successfully performed.

The `qr` dictionary consolidates the exam metadata used in the subsequent processing steps. Its main fields are:

- **`date`:** temporal identifier of the exam, consisting of the generation date and an internal MCTest timestamp.
- **`idClassroom`, `idExam`, and `idStudent`:** identifiers for the class, the exam, and the student.
- **`term`:** academic term.
- **`stylesheet`:** style sheet used in the exam form generation.
- **`var1` through `var5`:** number of questions at each difficulty level.
- **`text`:** number of essay questions.
- **`answer`:** number of answer choices per question.
- **`numquest`:** total number of questions.
- **`correct`** and **`dbtext`:** fields filled in during grading, containing the answer key and additional information.
- **`variations`** and **`variant`:** information about the exam versions.

These metadata identify the exam and the student, allowing the corresponding answer key to be selected and the subsequent steps of reading and grading the responses to be parameterized.

```
myFlagArea, qr = CVMCTest.cvMCTest.getQRCode(img_inicial, countPage)
myFlagArea, qr
```

```
(True,
 {'date': '260208-1770403541363',
  'idClassroom': '955',
  'idExam': '810',
  'idStudent': '448898',
  'term': '0',
  'stylesheet': '1',
  'var1': '50',
  'var2': '0',
  'var3': '0',
  'var4': '0',
  'var5': '0',
  'text': '0',
  'answer': '5',
```

```
  'numquest': 50,
  'correct': '',
  'dbtext': '',
  'variations': '0',
  'variant': '0'})
```

These metadata allow the exam to be identified, the corresponding answer key to be retrieved, and the subsequent processing steps to be parameterized.

After decoding the *QRCode*, processing returns to the answer area to locate the frames containing the student's markings.

6.9.4 Location of Answer Boxes

The image produced by `getAnswerArea` contains the entire useful region of the sheet, including the header, where the *QRCode* is located, and the boxes intended for the answers. Since the reading of the markings uses only these boxes, the region corresponding to the header is discarded by cropping the image using `img_getAnswerArea[300:, :]`.

Figure 6.16 shows this region of interest. Although this crop is subsequently used by the `findSquares` function to locate the answer boxes, MCTest initially performs the *QRCode* reading, as it contains the necessary metadata to identify the exam and configure the subsequent processing steps.

```
img_getAnswerArea_aux = img_getAnswerArea[300:,:]
mm.show(img_getAnswerArea_aux)
```

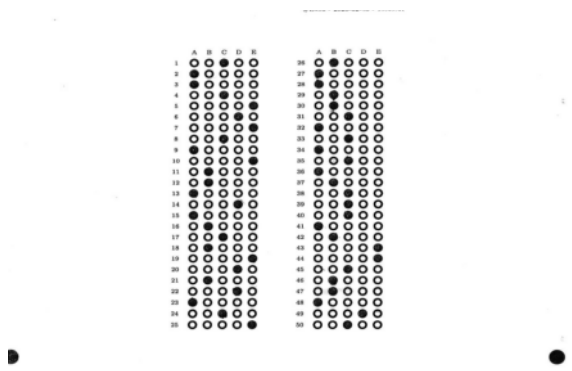


Figure 6.16: Lower crop of the response area, focusing on the bubble frames to be segmented.

The function `findSquares` receives the image of the answer area and the metadata stored in `qr`, returning the coordinates of the frames that delimit the question groups.

Each element of `rectSquares` contains the coordinates of the top-left and bottom-right vertices of an answer frame, which will be used in the bubble segmentation step.

```
rectSquares = CVMCTest.cvMCTest.findSquares(qr,img_getAnswerArea, countPage)
rectSquares
```

```
[[[np.int64(395), np.int64(350)], [np.int64(950), np.int64(516)]],
 [[np.int64(394), np.int64(602)], [np.int64(951), np.int64(767)]]]
```

6.9.5 Automatic Reading of Answers

Once the exam metadata (`qr`) and the coordinates of the answer boxes (`rectSquares`) are known, MCTest automatically identifies the alternatives marked by the student.

The following code integrates the steps presented earlier. For each box delimited in `rectSquares`, the functions `setColumns` and `setLines` estimate, respectively, the number of alternatives per question and the number of questions based on the spatial distribution of the bubbles. Then, `segmentAnswers` determines the alternative selected in each question, and `setAnswersOneLine` consolidates the results from all boxes into the field `qr['answers']`.

In the operating mode adopted in this chapter, in which MCTest runs independently of its database, the content of `qr['answers']` is compared to the answer key stored on the first page of the PDF file, corresponding to the exam template without statements used in the examples.

Figure 6.17 presents the answer boxes processed by the algorithm, while the program output displays the final content of `qr['answers']`.

```
testAnswers = []
if myFlagArea:

  imgQ_all = []

  for countSquare in range(len(rectSquares)):
      p1, p2 = rectSquares[countSquare]

      if True:
          imgQi = CVMCTest.cvMCTest.imgAnswers[p1[0]:p2[0], p1[1]:p2[1]]
          [NUM_COLUMNS, img] = CVMCTest.cvMCTest.setColumns(imgQi, countPage, countSquare)
          [NUM_LINES, img] = CVMCTest.cvMCTest.setLines(imgQi, countPage, countSquare)
          NUM_RESPOSTAS = NUM_COLUMNS
          NUM_QUESTOES = NUM_LINES

      imgQiNC = CVMCTest.cvMCTest.imgAnswers[p1[0]:p2[0], p1[1]:p2[1]]
      testAnswers.append(CVMCTest.cvMCTest.segmentAnswers(
          [imgQi, imgQiNC], countPage, countSquare, NUM_QUESTOES, qr

      ))

      imgQ_all.append(imgQiNC)

  qr = CVMCTest.cvMCTest.setAnswarsOneLine(testAnswers, qr)
  # leaves the answers of each frame on one line

mm.show(imgQ_all)
print(f"Answers read from the {len(qr['answers'].split(","))} questions: \
      \n{qr['answers'][:-19]}...")
```

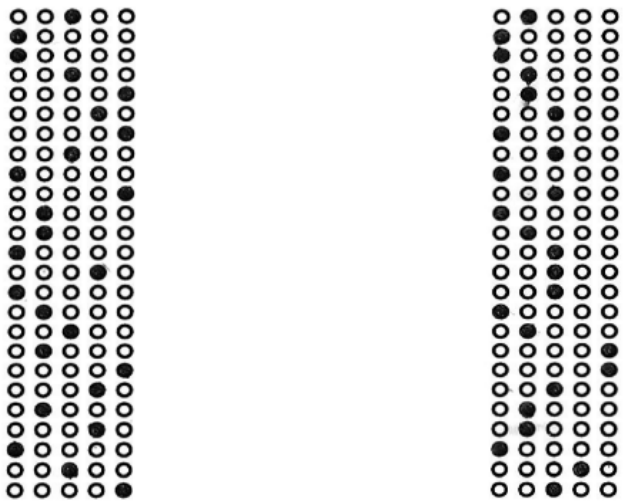


Figure 6.17: Answers read automatically by MCTest after segmentation and classification of all bubbles.

```
Answers read from the 50 questions:
C,A,A,C,E,D,E,C,A,E,B,B,A,D,A,B,C,B,E,D,B,D,A,C,E,B,A,A,B,B,C,A,C,A,C,A,B,C,C,C,...
```

Connecting the Dots

The field `qr['answers']` represents the final outcome of the Computer Vision pipeline presented in this chapter. Its computation integrates all the stages studied, from document rasterization and geometric rectification to the extraction of the answer area, QRCode decoding, frame localization, and identification of marked alternatives.
This pipeline illustrates the transition from a prototype to a production system. In MCTest, the Computer Vision algorithms remain essentially the same; the primary differences focus on software engineering aspects, such as exception handling, support for different form models, database integration, web interface, and auditing and maintenance mechanisms.
In the experiments of this chapter, the first page of the PDF file contains the answer key, while subsequent pages correspond to students' answer sheets. After obtaining `qr['answers']`, MCTest automatically compares the read responses with the answer key to calculate each student's score.
In the full system usage, grading results are consolidated into a CSV file and sent to the instructor along with a compressed file containing auxiliary information for auditing. Among these files are cropped images of questions where multiple markings or other situations requiring manual review were detected. The instructor can then inspect these images, decide on the most appropriate interpretation, and, if necessary, update the CSV file before the final import of grades.

6.10 Automated Industrial Inspection

Automated visual inspection is an application of Computer Vision in which images of parts or products are analyzed to verify compliance with previously defined quality criteria. On a production line, images can be acquired by cameras or other capture devices and processed automatically to identify defects, measure dimensions, or verify the presence of components.

The inspection strategy depends on the product characteristics, the type of defect of interest, and the availability of a reference image. This chapter presents two classical approaches:

- **Image subtraction:** compares the image of the inspected part with a reference image considered free of defects. Regions where the intensity difference exceeds a threshold are classified as possible defects. This approach assumes that the images are geometrically aligned and have been acquired under similar lighting conditions.

- **Texture analysis:** uses surface texture characteristics to identify regions whose appearance differs from the expected pattern, without the need for a reference image. This approach is suitable for materials with approximately homogeneous texture, such as fabrics, papers, and metallic surfaces.

In the following sections, these two strategies are illustrated through examples built from images in the `skimage.data` library. The goal is to present the working principles of each approach in experiments that can be fully reproduced by the reader.

6.10.1 References and Public Datasets

In industrial inspection applications, the performance of defect detection algorithms is often evaluated on public datasets, which provide representative images and, in many cases, ground truth annotations. In this chapter, however, the examples use synthetic images derived from `skimage.data` (see Figure 6.18), allowing all experiments to be reproduced without reliance on external databases.

For more comprehensive studies and algorithm comparisons, the following public datasets stand out:

- **MVTec Anomaly Detection Dataset (MVTec AD):** a collection of images of objects and textures, containing both defect-free and defective samples, accompanied by pixel-level segmentation masks for anomalous images (BERGMANN, 2019). Available at: https://www.mvtec.com/company/research/datasets/mvtec-ad.

- **Kolektor Surface-Defect Dataset (KolektorSDD):** a collection of images of industrial components with annotated surface defects, used in defect detection and segmentation studies (TABERNIK, 2020). Available at: https://www.vicos.si/resources/kolektorsdd/.

- **NEU Surface Defect Database:** a collection of images of rolled steel surfaces, organized into six categories of surface defects, frequently employed in the evaluation of classification and detection methods (SONG, 2013). Available at: http://faculty.neu.edu.cn/songkechen/zh_CN/zdylm/263270/list/index.htm.

```python
import numpy as np
from skimage import data, color
from morph import mm

# Reference image (defect-free product)
product_color = data.coffee()
product_gray = color.rgb2gray(product_color)

# Insertion of simulated defect: dark scratch of 10×100 px
defect_image = np.copy(product_gray)
defect_image[100:110, 200:300] = 0.1

# Detection by subtraction and thresholding
difference = np.abs(product_gray - defect_image)
defect_threshold = 0.15          # adjustable according to the application
detected_defect = (difference > defect_threshold).astype(np.uint8) * 255

mm.show(
    [product_gray, defect_image, detected_defect],
    titles=["Reference", "With defect", "Detected defect"],
    cols=3,
    figsize=(12, 4)
)

status = "Defeito detectado." if detected_defect.any() else "Produto conforme."
print(status)
```

Figure 6.18: Defect detection by image subtraction: reference product, image with simulated defect, and detected anomaly mask.

```
Defeito detectado.
```

Image Subtraction: Geometric Registration and Operating Principle

Image subtraction assumes that the inspection image is geometrically aligned with the reference image. Differences in positioning, rotation, scale, or perspective produce regions of difference that may be mistaken for defects.
In applications with controlled acquisition, this alignment is achieved during capture using mechanical fixtures, conveyors, and fixed cameras, allowing successive images to be compared directly. One example is the inspection of a tool box always positioned in the same orientation to verify that no item is missing.
When such control is not possible, **image registration** is employed, which estimates a geometric transformation to compensate for differences in translation, rotation, scale, and, when necessary, perspective.
After registration, a pixel-by-pixel comparison between the two images is performed. In regions without changes, intensity differences tend to be close to zero; where a defect exists, local differences emerge that can be highlighted by thresholding. The use of the absolute difference allows detecting both defects that are lighter and darker than the reference.
The performance of the method depends mainly on the quality of geometric alignment and on the choice of the threshold used to separate small acquisition variations from differences associated with defects.
Figure 6.19 illustrates the effect of misalignment between images and the importance of geometric registration before applying subtraction.

6.10.2 Defect Detection by Texture Analysis

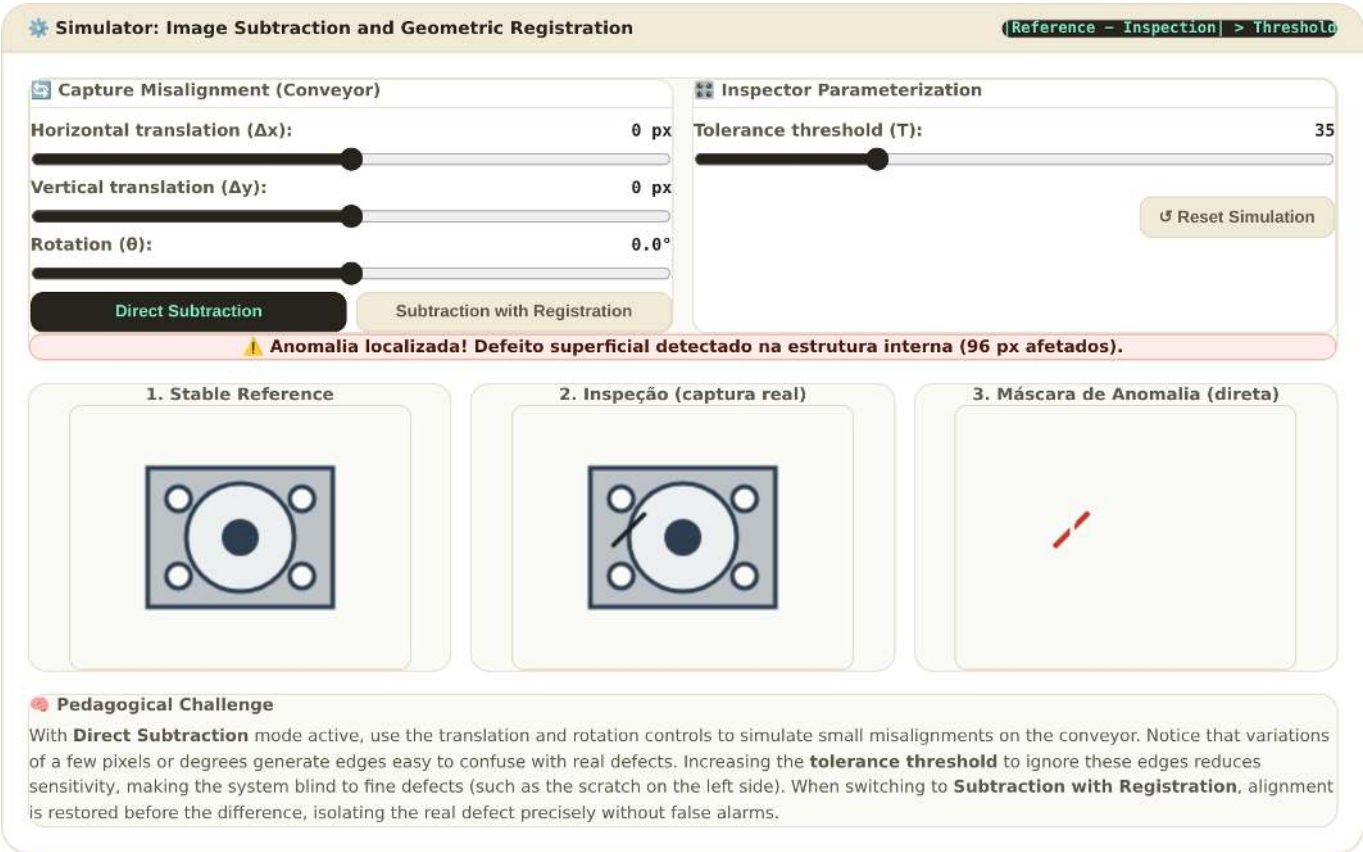
In applications where there is no reference image, defect detection can be based on the surface texture characteristics. In this case, the goal is to identify regions whose appearance differs from the predominant pattern of the material.

In this example, the **local variance** is used as a measure of heterogeneity. For each image position, the variance of intensity levels is calculated within a neighborhood of fixed dimensions. Regions with low variance tend to exhibit a more uniform texture, while local alterations, such as scratches, stains, or imperfections, may produce higher values of this measure.

Figure 6.20 illustrates this procedure using the image `skimage.data.brick()`. Initially, the local variance map is computed using a sliding window. Subsequently, thresholding is applied to highlight regions whose variance exceeds the specified value, identifying potential areas of interest for inspection.

Mathematical Modeling

Consider a grayscale image represented by



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap06/sim-06-industrial.html

Figure 6.19: Interactive industrial inspection simulator by image subtraction: control the geometric misalignment distortions (registration) and the detection threshold to observe the impact on false positives.

$$f : \Omega \subset \mathbb{Z}^2 \to \mathbb{R},$$

where Ω is the image domain and $f(x, y)$ represents the pixel intensity at coordinates (x, y). In 8-bit images, these intensities belong to the interval $[0, 255]$. In this example, however, they were normalized to the interval $[0, 1]$, without altering the algorithm's operation.

For each position in the image, a square neighborhood $W_{x,y}$ of dimensions 15×15 pixels is considered.

The local mean is given by

$$\mu(x, y) = \frac{1}{|W_{x,y}|} \sum_{(u,v)\in W_{x,y}} f(u, v),$$

and the local variance is computed as

$$\sigma^2(x, y) = \frac{1}{|W_{x,y}|} \sum_{(u,v)\in W_{x,y}} f(u, v)^2 - \mu(x, y)^2.$$

In the following implementation, these two means are obtained using the `cv2.blur` function,

```
mean  = cv2.blur(img, (15,15))
mean2 = cv2.blur(img**2, (15,15))
var   = mean2 - mean**2
```

Next, the difference between the variance maps of the reference image and the inspected image is computed,

$$D(x,y) = \left|\sigma_d^2(x,y) - \sigma_r^2(x,y)\right|,$$

where $\sigma_r^2(x,y)$ and $\sigma_d^2(x,y)$ are the local variances of the reference image and the image containing the defect, respectively. After normalizing the map $D(x,y)$, a thresholding operation is applied to obtain the mask of possible anomalies.

```
import numpy as np
import cv2
from skimage import data as skdata
from morph import mm

# Uniform texture image (brick)
texture = skdata.brick().astype(np.float32) / 255.0

# Insertion of synthetic defect: light spot 20×80 px
texture_defect = np.copy(texture)
texture_defect[60:80, 80:160] = 0.95

# Local variance map (window 15×15)
def variancia_local(img, ksize=15):
    img_f = img.astype(np.float32)
    mean  = cv2.blur(img_f, (ksize, ksize))
    mean2 = cv2.blur(img_f ** 2, (ksize, ksize))
    return np.clip(mean2 - mean ** 2, 0, None)

var_ref    = variancia_local(texture)
var_defect = variancia_local(texture_defect)
diff_var   = np.abs(var_defect - var_ref)

# Normalize and threshold
diff_norm = (diff_var / diff_var.max() * 255).astype(np.uint8)
_, mask   = cv2.threshold(diff_norm, 30, 255, cv2.THRESH_BINARY)

mm.show(
    [texture, texture_defect, diff_norm, mask],
    titles=["Original texture", "With defect", "Δ local variance", "Detected anomaly"],
    cols=4,
    figsize=(16, 4)
)

status = "Defeito de textura detectado." if mask.any() else "Superfície conforme."
print(status)
```

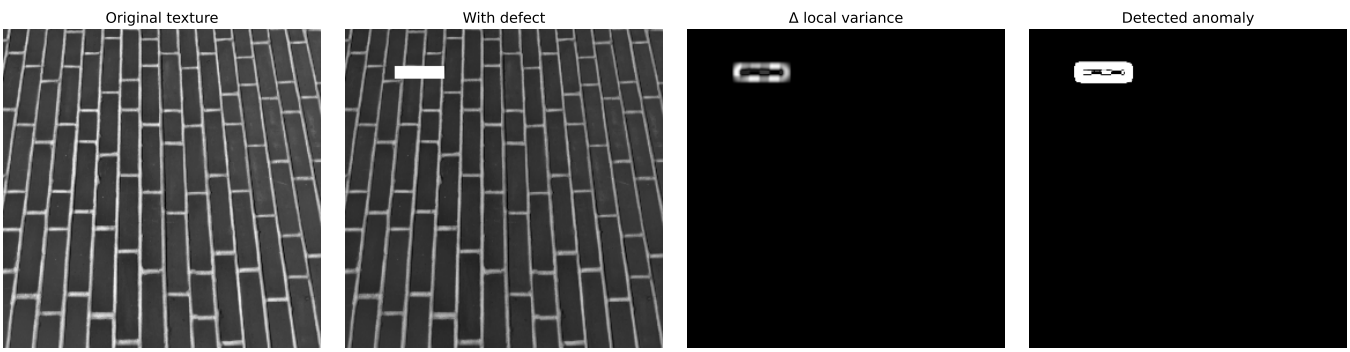


Figure 6.20: Detection of texture heterogeneity: local variance map and anomaly mask.

```
Defeito de textura detectado.
```

ℹ 🧠 Why It Works — Texture Analysis

Local variance measures the dispersion of intensities within a neighborhood of the image. In regions where the texture remains uniform, this measure tends to vary little. When a defect modifies the surface pattern, the distribution of intensities also changes, producing differences in local variance.
In this chapter, detection is carried out by comparing the variance maps of the reference image and the defective image. After normalization, thresholding is applied to highlight regions where this difference exceeds a specified value.
The main parameters of the method are the window size used in the variance calculation and the threshold employed in segmentation. Smaller windows are more sensitive to fine details, while larger windows produce smoother maps and may reduce the response to small-sized defects.

6.11 Summary

This chapter presented Computer Vision methods applied to document analysis and automated visual inspection. The main techniques studied were:

- **Document preprocessing:** application of background normalization, adaptive equalization (CLAHE), and Otsu thresholding to reduce the effects of non-uniform illumination and improve text segmentation.
- **Optical character recognition (OCR):** conversion of document images into encoded text using Tesseract OCR, highlighting the influence of preprocessing on recognition quality.
- **Machine translation:** application of natural language processing techniques to translate the text obtained by OCR.
- **Geometric document rectification:** use of the Canny edge detector, the Hough Transform, and projective transformations to correct the perspective of scanned documents.
- **Form localization and rectification:** use of morphological operations, contour analysis, and perspective transformation to identify reference markers and automatically extract regions of interest.
- **Reading of two-dimensional and one-dimensional codes:** detection and decoding of *QRCodes* and barcodes for automatic identification of documents and metadata.
- **Optical mark recognition (OMR):** automated reading of forms and answer sheets, illustrated by a case study of the MCTest system.
- **Industrial inspection:** defect detection by comparison with a reference image and by analysis of local variance.

Throughout the chapter, the algorithms were implemented and evaluated with images from the `skimage.data` library and with real documents, allowing the reproduction of the presented experiments.

Next Steps

The methods presented in this chapter show how Digital Image Processing, Computer Vision, and Natural Language Processing techniques can be integrated into *pipelines* for automated document analysis.

The next chapter will study **feature extraction** and **pattern recognition** techniques, with emphasis on descriptors capable of representing images by numerical attributes for comparison, classification, and automatic recognition. These concepts form the basis for the chapters dedicated to machine learning and deep learning applied to Computer Vision.

6.12 🤖 Using Gemini Notebook as a Complementary Tutor

As a study aid for this chapter, the use of **Gemini Notebook** as a complementary tutor is recommended. The tool employs artificial intelligence models to answer questions, prepare summaries, and explain concepts

based on the documents provided as reference sources, allowing students to review the content interactively.

🎓 Study with the Intelligent Tutor

🚀 ACCESS Gemini Notebook: CHAPTER 06

🌐 **Language and Programming Language**

The project for this chapter in Gemini Notebook was built solely with the text in **Portuguese** and the code examples in **Python**. If you are studying from the English or French edition, or following the C++ track, the tutor's responses may not correspond exactly to the version you are reading.

⚠️ **Critical Use of Responses**

The responses generated by Gemini Notebook are produced automatically by an artificial intelligence model and may contain omissions or inaccuracies. For this reason, they should be used as supporting material, not as a substitute for studying the chapter.
Whenever questions arise, consult the text of this book, run the examples provided, and, when necessary, supplement your research with books, scientific articles, and other reliable academic sources.

6.13 Exercise List

The following exercises consolidate the concepts presented in this chapter through adaptations, experiments, and extensions of the algorithms developed throughout the text.

1. **(10%)** Investigate the influence of the inclination angle in the *deskew* stage. Generate rotated versions of the `skimage.data.page()` image for angles between -10° and 10°, apply the algorithm presented in the chapter, and compare the estimated angle with the angle used in the rotation. Present the results in a table and discuss the accuracy of the method.

2. **(15%)** Apply background normalization and CLAHE (using at least three combinations of `clipLimit` and `tileGridSize`) to the `skimage.data.page()` image artificially degraded with an illumination gradient and lateral shadow. Segment each version using Otsu's method and compare the results using the number of spurious connected components and the IoU metric relative to a manually constructed reference mask.

3. **(15%)** Investigate the sensitivity of circularity filtering, $C = \frac{4\pi A}{P^2}$, in detecting circular markers. Using the simulator from Figure 6.9, generate synthetic disks with increasing geometric noise and evaluate the thresholds $C \in \{0.5,\ 0.6,\ 0.7,\ 0.8\}$. Present a table relating the threshold to the number of false positives and false negatives and discuss the trade-off between sensitivity and specificity.

4. **(15%)** From the four detected markers, implement perspective rectification using `cv2.getPerspectiveTransform` and `cv2.warpPerspective`. Then, artificially perturb the coordinates of the control points with Gaussian noise of standard deviation $\sigma \in \{1, 3, 5\}$ pixels and evaluate the reprojection error obtained after the inverse homography.

5. **(15%)** Adapt the acquisition and rectification *pipeline* developed in this chapter to process documents containing linear barcodes instead of *QRCodes*. Rasterize the PDF with `pdf2image` at 300 DPI, apply *deskew*, decode the symbol with `pyzbar`, and present the rectified image along with the obtained character sequence.

6. **(15%)** Extend the MCTest bubble reading to identify three situations: **OK**, **BLANK** (no alternative marked), and **DOUBLE MARKING** (two or more alternatives above a fill threshold). Evaluate at least three values of this threshold, present the results in a `pandas.DataFrame`, and discuss its influence on the classification of responses.

7. **(15%)** Build an industrial inspection *pipeline* combining image subtraction and local texture variance

analysis on a set of synthetic images containing simulated defects. For each image, generate a reference mask (*ground truth*), compute the IoU metric (*Intersection over Union*) of both approaches for different decision thresholds, and present the results in tables and visualizations produced with `mm.show`.

8. **(Bonus – 10%)** Manually implement the inclination angle estimation without using `cv2.HoughLines` or `cv2.HoughLinesP`. From the edge map obtained by the Canny detector, construct the Hough Transform accumulator, $\rho = x\cos\theta + y\sin\theta$, for $\theta \in [-45°, 45°]$, identify the accumulator maxima, and estimate the inclination by the median of the detected lines. Compare the results with the OpenCV implementation and discuss the influence of spurious lines on the final estimate.

Chapter References

The theoretical foundation and case studies presented in this chapter are supported by the following references:

- Gonzalez (2018), for the fundamentals of edge detection, thresholding, segmentation, morphological operations, optical character recognition, and geometric transformations applied to document analysis.
- Szeliski (2022), for the Hough Transform, image registration and alignment, projective transformations (homographies), and the fundamentals of automated visual inspection.
- Bradski (2008), for the use of the OpenCV library in the stages of edge detection, Hough Transform, geometric transformations, contour analysis, and QR Code decoding.
- Smith (2007) and Smith (2013), for the architecture, operation, and evolution of the **Tesseract OCR** optical character recognition engine, used in the OCR examples presented in this chapter.
- Bahdanau (2015) and Vaswani (2017), for the fundamentals of neural network-based machine translation, including attention mechanisms and *transformer* architectures.
- Zampirolli (2023), for the description of the MCTest system, used as a case study of a complete pipeline for automated reading and grading of answer sheets.
- Bergmann (2019), Tabernik (2020), and Song (2013), for the public industrial inspection datasets **MVTec AD**, **KolektorSDD**, and **NEU Surface Defect Database**, used as a reference for evaluating and comparing defect detection algorithms.

Run Colab | Open GitHub

6.14 💻 Practical Section with Programming Exercises

The programming exercises (EPs) in this section complement the concepts presented throughout Chapter 6 by implementing algorithms related to industrial inspection and document analysis. The goal is to consolidate the fundamentals studied by reproducing, on a reduced scale, steps of a typical Computer Vision pipeline.

Unlike the previous chapters, whose exercises emphasized operations more directly related to image data, the EPs in this chapter focus on the **intermediate quantities** produced during processing, such as areas, perimeters, circularity, line angles, bubble fill degrees, variance maps, and difference maps. This approach allows each stage of the pipeline to be understood and validated independently, without relying on specialized libraries for image acquisition, marker detection, or code decoding—except for the chapter's closing exercise (EP06_08), which intentionally introduces the use of OpenCV for the segmentation and actual decoding of a QRCode, closing the loop between theoretical concepts and the tools employed in practice.

The exercises follow the same conceptual sequence as the chapter, in increasing order of complexity. Initially, segmentation evaluation metrics are addressed, used to quantify the quality of binary masks. Next, geometric criteria for marker selection, classification of markings on forms, and estimation of document skew through the Hough Transform are studied. In the final part, the exercises explore illumination normalization, defect

detection through texture analysis, and the integration of geometric registration and image subtraction in a simplified industrial inspection pipeline.

Each exercise represents an isolated stage of a real Computer Vision system, allowing individual validation of concepts that, in industrial applications, are combined into a single inspection pipeline.

Difficulty Legend

Level	Meaning	EPs
🟢	Very easy / easy — implementation of a single concept or simple algorithm	EP06_01, EP06_02
🟡	Easy–medium — handling multiple cases or using simple statistical criteria	EP06_03, EP06_04
🟠	Medium — pointwise matrix processing	EP06_05
🔴	Difficult — matrix processing with neighborhood operations (sliding window)	EP06_06
🟣	Very difficult — integration of multiple stages of a Computer Vision pipeline	EP06_07
⚫	Special — use of specialized library (`cv2`) for geometric segmentation and actual barcode/QRCode decoding	EP06_08

❗ Guidelines for Solving the Programming Exercises

Unless otherwise stated, all exercises use the matrix coordinate convention `[row][column]`, with the origin at $(0, 0)$ in the upper-left corner of the image.
When numerical rounding is required, standard rounding to the nearest integer should be used (*round half away from zero*, with `np.floor(img + 0.5)`). Comparisons with thresholds (for example, circularity, variance, intensity difference, or fill degree) should be considered **strict** (`>`), unless the statement explicitly specifies another criterion.
Each exercise was designed to emphasize a specific concept presented in the chapter. It is recommended to initially implement the solution directly and, only after its validation, seek more efficient or more general alternatives.

🎯 Objective of this Notebook

This notebook was designed to support the development, validation, and testing of solutions for the **Programming Exercises (PEs)** in an interactive environment, such as Google Colab or Jupyter Notebook. After verifying that the implementation works with the provided test cases, the code can be submitted to Moodle for official evaluation.

Download

Run the following cell to obtain the files `morph.py` and `testsuite.py`, used by the exercises in this chapter.

```python
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup(testsuite=True)
from morph import mm
from testsuite import TestSuite
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Running the Tests

After implementing the solution, run `TestSuite("EP06_01.extension").run()` in a new cell, replacing `extension` with the language used (`.py`, `.java`, `.c`, `.cpp`, `.js`, or `.r`). The system automatically retrieves the test cases from the course repository, runs the program, and displays the evaluation result.

In Python, it is also possible to test the solution directly from a *string*, without the need to save the code to a file. To do this, store the program in a variable and use the `run_code` method:

```python
codigo = """
# ... your code here ...
"""

TestSuite("EP06_01").run_code(codigo)
```

6.14.1 EP06_01 🟢 Segmentation Evaluation by IoU (*Intersection over Union*)

Throughout this chapter, several stages of the *pipeline* produce **binary masks**, such as in document segmentation, *QRCode* localization, and defect detection. To objectively evaluate the quality of these segmentations, it is necessary to compare them with a reference mask (*ground truth*).

One of the most widely used metrics for this purpose is the **IoU** (*Intersection over Union*), defined as the ratio between the intersection area and the union area of two binary masks. The higher the IoU value, the greater the agreement between the segmentation produced by the algorithm and the reference.

6.14.1.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (number of rows) and C (number of columns).
2. **Reference mask:** Read the $L \times C$ binary elements (0 or 1) of the `ref` matrix.
3. **Predicted mask:** Read the $L \times C$ binary elements (0 or 1) of the `pred` matrix.
4. **Intersection:** Count the number of positions (i, j) for which `ref[i][j] = 1` and `pred[i][j] = 1`.
5. **Union:** Count the number of positions (i, j) for which `ref[i][j] = 1` or `pred[i][j] = 1`.
6. **Degenerate case:** If the union equals 0, set IoU = 1,0, since both masks are empty.
7. **Calculation:** If the union is greater than zero, compute

$$\text{IoU} = \frac{|\text{Intersecao}|}{|\text{Uniao}|}.$$

8. **Classification:** Determine the qualitative classification using the IoU value **before** rounding.
9. **Rounding:** Display the IoU with four decimal places.
10. **Output:** Print, in this order, the intersection, union, IoU, and classification.

6.14.1.2 📌 Computational Restrictions

- If the union equals 0, the division must not be performed; the IoU must be defined as 1,0.

- The classification ranges use non-strict comparisons ($\geq$).
- The classification must be performed using the IoU value in full precision, before rounding for display.

6.14.1.3 🧠 Theoretical Foundation

The IoU is defined by

$$\text{IoU} = \frac{|R \cap P|}{|R \cup P|},$$

where:

- R represents the set of pixels belonging to the reference mask;
- P represents the set of pixels belonging to the predicted mask;
- $|R \cap P|$ corresponds to the number of pixels belonging simultaneously to both masks;
- $|R \cup P|$ corresponds to the number of pixels belonging to at least one of the masks.

IoU Range	Classification	Interpretation
IoU $\geq$ 0,90	`EXCELENTE`	Very high agreement between the masks.
0,70 $\leq$ IoU $<$ 0,90	`BOM`	Small differences between the masks.
0,50 $\leq$ IoU $<$ 0,70	`ACEITAVEL`	Partial agreement between the masks.
IoU $<$ 0,50	`RUIM`	Low agreement between the masks.

The IoU depends only on the overlap between the masks and is therefore independent of the image size.

6.14.1.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: integer L.
- Line 2: integer C.
- Next L lines: binary elements (0 or 1) of the `ref` matrix.
- Next L lines: binary elements (0 or 1) of the `pred` matrix.

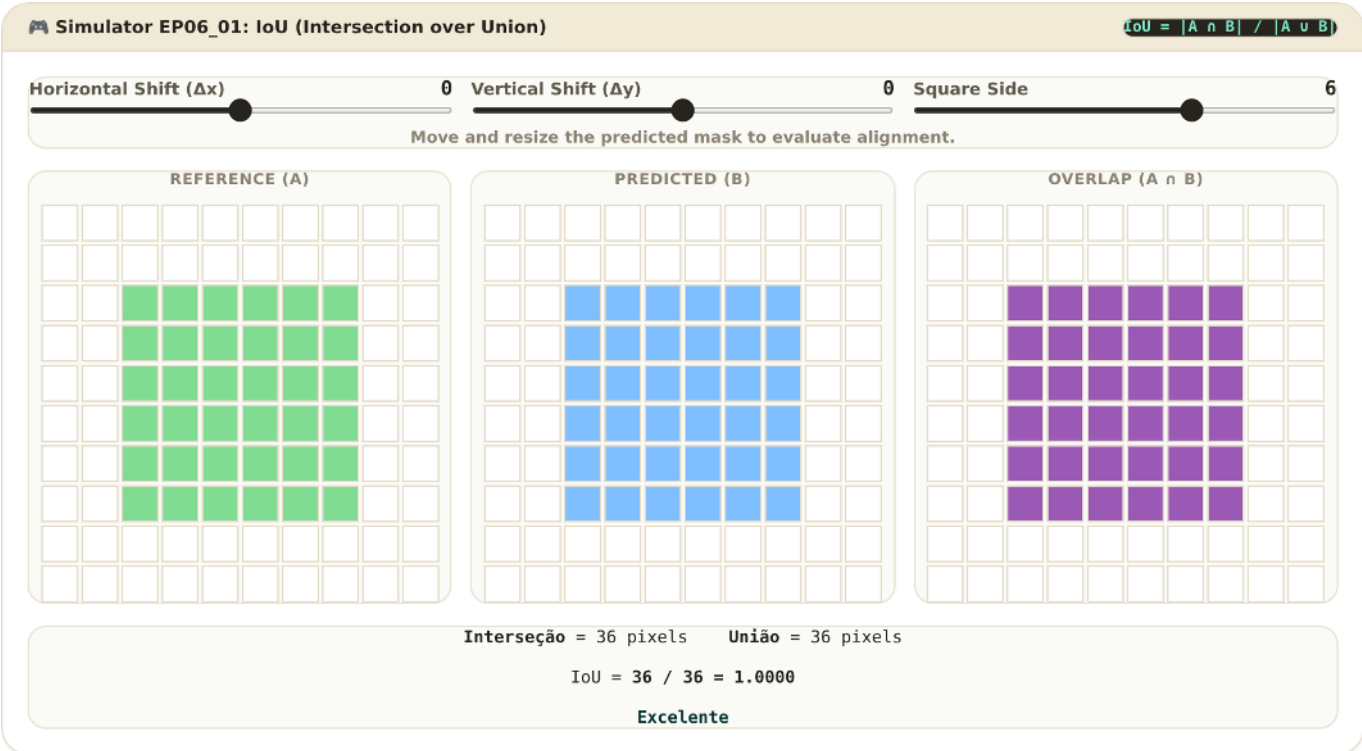
Output:

- Line 1: `Intersecao: X`
- Line 2: `Uniao: Y`
- Line 3: `IoU: Z`
- Line 4: `Classificacao: NOME`

The `IoU` value must be printed with four decimal places.

6.14.1.5 📌 Examples

Input	Output	Observation
2 2 1 1 0 0 1 0 0 0	Intersecao: 1 Uniao: 2 IoU: 0.5000 Classificacao: ACEITAVEL	Half of the reference region was correctly segmented.
2 2 0 0 0 0 0 0 0 0	Intersecao: 0 Uniao: 0 IoU: 1.0000 Classificacao: EXCELENTE	Both masks are empty; by convention, IoU = 1,0.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap06/sim-ep0601.html

Figure 6.21: EP06_01 Simulator: IoU between reference mask and predicted mask

```
%%writefile EP06_01.py
# Python code
```

```
Overwriting EP06_01.py
```

```
TestSuite("EP06_01.py").run()
```

```
<IPython.core.display.HTML object>
```

6.14.2 EP06_02 🟢 Marker Filtering by Circularity

After image segmentation, it is common for several connected components to be identified. In applications such as document rectification, only a few of these components correspond to the reference markers used for

image alignment. A criterion often employed to select these markers is **circularity**, which measures how close a component's shape is to a circle.

In this exercise, each component is described by its area A and its perimeter P. The goal is to compute its circularity and decide, based on a provided threshold, whether the component should be accepted or rejected as a marker candidate.

6.14.2.1 📋 Implementation Guidelines

1. **Quantity:** Read the integer N (number of candidates) and the circularity threshold C_{limiar} (real number).
2. **Candidate data:** For each of the N candidates, read the area A (integer) and the perimeter P (real number).
3. **Circularity:** Compute $C = \frac{4\pi A}{P^2}$, where:

- A is the component's area;
- P is the component's perimeter;
- C is the circularity.

4. **Degenerate case:** If $P = 0$, consider $C = 0$ and directly classify the candidate as `REJECTED`.
5. **Classification:** If $C > C_{\text{limiar}}$, classify the candidate as `ACCEPTED`; otherwise, classify it as `REJECTED`.
6. **Rounding:** Display the value of C with four decimal places.
7. **Output:** For each candidate, print the value of C followed by the classification. At the end, print the total number of accepted candidates.

6.14.2.2 📌 Computational Constraints

- Use the constant π from the language's standard library (e.g., `math.pi`), without approximations.
- The comparison must be performed using the full-precision value of C, before rounding for display.
- The acceptance criterion is strict ($C > C_{\text{limiar}}$).
- If $P = 0$, the division must not be performed.

6.14.2.3 🧠 Theoretical Background

Circularity is a geometric descriptor defined by $C = \frac{4\pi A}{P^2}$, where:

- A is the component's area;
- P is the component's perimeter;
- C is the circularity.

For a perfect circle, $C = 1$. As the shape becomes more elongated or irregular, the perimeter grows faster than the area, reducing the value of C.

Shape	Approximate circularity	Interpretation
Circle	1,0000	Circular shape.
Square	0,7854	Approximately compact shape.
Elongated or irregular shape	$C \ll 1$	Low circularity.
$P = 0$	0 (adopted convention)	Degenerate contour.

Circularity is invariant to translation, rotation, and scaling, and it is widely used to distinguish approximately circular components from other shapes.

6.14.2.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: integer N.

- Line 2: real number C_{limiar}.
- Next N lines: area A (integer) and perimeter P (real), separated by a space.

Output:

- One line for each candidate, in the format `C ACCEPTED` or `C REJECTED`, with C presented with four decimal places.
- Last line: `Total accepted: X`.

6.14.2.5 📌 Examples

Input	Output	Observation
3 0.6 78 31.4 100 40 50 60	0.9941 ACCEPTED 0.7854 ACCEPTED 0.1745 REJECTED Total accepted: 2	Approximately circular candidate, compact shape, and elongated shape.
1 0.9 10 0	0.0000 REJECTED Total accepted: 0	Null perimeter: degenerate contour.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap06/sim-ep0602.html

Figure 6.22: Simulator EP06_02: Marker Filter by Circularity

```
%%writefile EP06_02.py
# Python code
```

```
Overwriting EP06_02.py
```

```
TestSuite("EP06_02.py").run()
```

```
<IPython.core.display.HTML object>
```

6.14.3 EP06_03 🟡 Classification of Markings on Answer Sheets (OMR)

After sheet rectification and segmentation of answer frames, MCTest estimates, for each bubble, a **fill degree**, represented by a value between 0 and 100. Based on these values, the system must automatically determine the marked alternative, also identifying blank questions and cases of multiple markings.

In this exercise, you will implement this decision stage of the OMR *pipeline*. The classification depends on a fill threshold: small variations in this value can alter the result of automatic reading.

6.14.3.1 📋 Implementation Guidelines

1. **Parameters:** Read integers Q (number of questions) and K (number of alternatives per question, with $2 \leq K \leq 26$) and the fill threshold Th (a real number between 0 and 100).
2. **Fill degrees:** For each of the Q questions, read the K real values corresponding to alternatives `A`, `B`, `C`, …, in input order.
3. **Counting markings:** For each question, count how many alternatives have a fill degree **strictly greater** than Th.
4. **Classification:**
   - If no alternative exceeds Th, classify the question as `BRANCO`.
   - If exactly one alternative exceeds Th, print the corresponding letter (`A`, `B`, `C`, …).
   - If two or more alternatives exceed Th, classify the question as `DUPLA_MARCACAO`.
5. **Output per question:** Print, in reading order, the classification of each question.
6. **Totals:** Finally, print the number of questions `OK` (single marking), `BRANCO`, and `DUPLA_MARCACAO`.

6.14.3.2 📌 Computational Constraints

- **Strict comparison:** only values greater than Th are considered valid markings; values exactly equal to the threshold must not be counted.
- **Alternative letters:** index 0 corresponds to alternative `A`, index 1 to alternative `B`, and so on.
- **Multiple markings:** whenever two or more alternatives exceed the threshold, the classification must be `DUPLA_MARCACAO`, regardless of their respective fill degrees.

6.14.3.3 🧠 Theoretical Foundation

Situation	Classification	Interpretation
Exactly one alternative above the threshold	Letter of the alternative	Valid answer
No alternative above the threshold	`BRANCO`	Unanswered question
Two or more alternatives above the threshold	`DUPLA_MARCACAO`	Ambiguous answer

The fill threshold controls the sensitivity of the algorithm. Very low values tend to increase the number of `DUPLA_MARCACAO`, while very high values may increase the number of questions classified as `BRANCO`.

6.14.3.4 📦 Input and Output Specification (VPL)

Input:

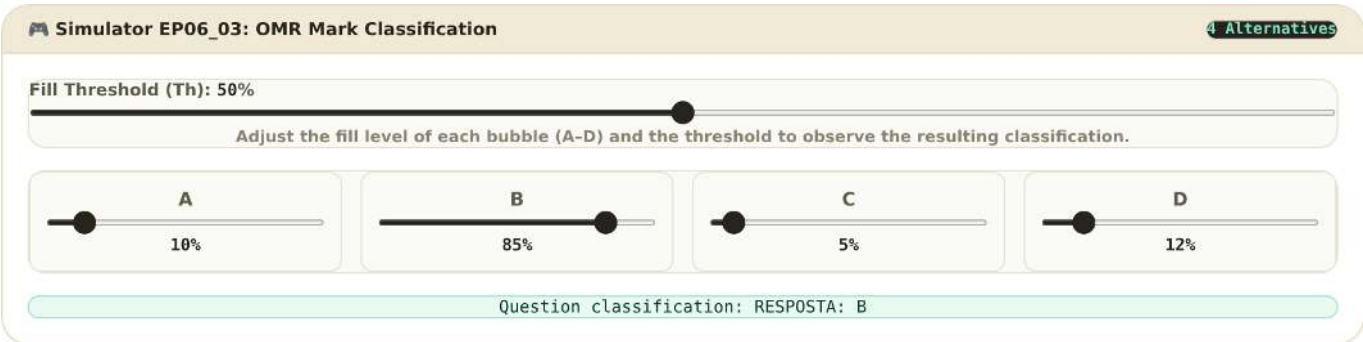
- Line 1: Integer Q.
- Line 2: Integer K.
- Line 3: Real number Th.
- Next Q lines: K real numbers, corresponding to the fill degrees of the alternatives.
- Line 1: Integer Q and K.

Output:

- Q lines, each containing the classification of the respective question.
- Final line: `OK: x  BRANCO: y  DUPLA_MARCACAO: z`.

6.14.3.5 📌 Examples

Input	Output	Observation
3 4 50 10 85 5 12 20 15 18 22 90 88 10 5	B BRANCO DUPLA_MARCACAO OK: 1 BRANCO: 1 DUPLA_MARCACAO: 1	In the first question only `B` exceeds the threshold; in the second, no alternative exceeds it; in the third, `A` and `B` exceed the threshold.
1 2 50.0 50 50	BRANCO OK: 0 BRANCO: 1 DUPLA_MARCACAO: 0	Values equal to the threshold are not considered valid markings.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap06/sim-ep0603.html

Figure 6.23: EP06_03 Simulator: OMR Mark Classification

```
%%writefile EP06_03.py
# Python code
```

```
Overwriting EP06_03.py
```

```
TestSuite("EP06_03.py").run()
```

```
<IPython.core.display.HTML object>
```

6.14.4 EP06_04 🟡 Median Angular Slope Estimator (*Deskew*)

After edge detection and the application of the Hough Transform, a set of candidate lines for the predominant orientation of the document is obtained. Each line provides an estimate of the skew angle, calculated by

$$\text{angle} = \text{rad2deg}(\theta) - 90.$$

However, not all lines correspond to document lines: some result from noise, shadows, or other image elements. In this exercise, you will implement the robust estimation step of the skew angle, filtering plausible values and computing their median.

6.14.4.1 📋 Implementation Guidelines

1. **Quantity:** Read the integer M, corresponding to the number of estimated angles.
2. **Angles:** Read the M real values, in degrees.
3. **Filtering:** Keep only the angles that **strictly** satisfy $-45 < \text{angle} < 45$.
4. **No candidates:** If no angle remains after filtering, print exactly `SEM_CORRECAO`.
5. **Median:** If valid angles exist:

- if the quantity is odd, the median is the central element of the ordered sequence;
- if it is even, the median is the arithmetic mean of the two central elements.

6. **Output:** Print the median rounded to two decimal places (standard rounding, *round half away from zero*, with `np.floor(img + 0.5)`).

6.14.4.2 📌 Computational Constraints

- **Open interval:** angles equal to -45 or 45 must not be considered.
- **Precision:** compute the median using the original values; rounding must be performed only at the output.
- **Empty case:** if there are no valid angles, no median must be computed.

6.14.4.3 🧠 Theoretical Background

Situation	Result
Most angles concentrated around the true skew	The median approximates the document orientation.
Few discrepant angles (*outliers*)	The median is little influenced by these values.
Angles outside the interval $(-45°, 45°)$	They are discarded before the computation.
No valid angle	No correction is applied (`SEM_CORRECAO`).

The median is used because it is more robust than the mean in the presence of a few discrepant values, producing a more stable estimate of the predominant document skew.

6.14.4.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer M.
- Line 2: M real numbers, corresponding to the angles in degrees.

Output:

- A single line containing the estimated angle, with two decimal places, or the word `SEM_CORRECAO` if no angle is valid.

6.14.4.5 📌 Examples

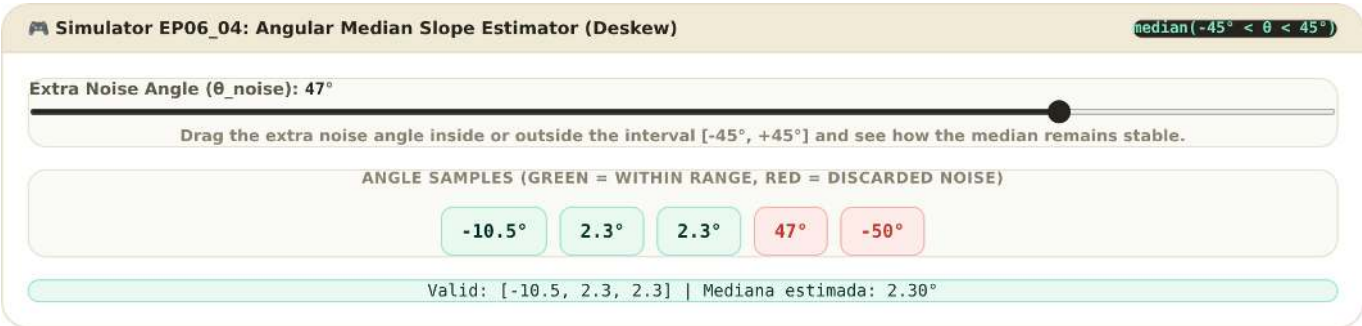
Input	Output	Remark
5 -50 -10.5 2.3 2.3 47	2.30	Only angles in the interval $(-45, 45)$ are considered; the median is 2.3.
4 -46 50 45 -45	SEM_CORRECAO	No angle belongs to the open interval $(-45, 45)$.

```
%%writefile EP06_04.py
# Python code
```

```
Overwriting EP06_04.py
```

```
TestSuite("EP06_04.py").run()
```

```
<IPython.core.display.HTML object>
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap06/sim-ep0604.html

Figure 6.24: EP06_04 Simulator: Slope Estimator by Angular Median

6.14.5 EP06_05 🟠 Background Normalization by Division (Illumination Correction)

A form was photographed under non-uniform illumination, causing one side of the page to appear lighter than the other. Under these conditions, global thresholding by Otsu can produce unsatisfactory results, since a single threshold does not properly separate text and background across the entire image. The solution presented in the chapter consists of **normalizing the background** by dividing the original image by a heavily smoothed version of itself, which represents the low-frequency illumination.

In this exercise, the original image and the smoothed background (equivalent to the result of a `cv2.GaussianBlur` with high σ) are already provided. Your task is to implement the normalization step that produces the corrected image.

6.14.5.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers L (rows) and C (columns).
2. **Original image:** Read the $L \times C$ integer values of the matrix `img` (intensities between 0 and 255).
3. **Estimated background:** Read the $L \times C$ integer values of the matrix `bg` (intensities between 0 and 255, always strictly greater than zero).
4. **Normalization:** For each position (i, j), compute

$$\text{value}(i, j) = \frac{\text{img}(i, j)}{\text{bg}(i, j)} \times 255.$$

5. **Rounding:** Round the result to the nearest integer (*round half away from zero*, using `np.floor(img + 0.5)`).
6. **Saturation:** Clip the obtained value to the interval $[0, 255]$.
7. **Output:** Print the resulting matrix `img_norm`.

6.14.5.2 📌 Computational Constraints

- **Division by zero:** the input guarantees $\text{bg}(i, j) > 0$ at all positions.
- **Order of operations:** first round, then apply saturation.
- **Independent processing:** each pixel must be normalized individually, without using information from neighboring pixels.

6.14.5.3 🧠 Theoretical Foundation

Situation	Effect of normalization
$\text{img}(i, j) = \text{bg}(i, j)$	Result equal to 255, corresponding to the normalized background.
$\text{img}(i, j) < \text{bg}(i, j)$	Result less than 255, preserving darker regions, such as text.

Situation	Effect of normalization
$\text{img}(i,j) > \text{bg}(i,j)$ Background with non-uniform illumination	Result greater than 255, subsequently saturated. The division reduces slow illumination variations, making the image more homogeneous.

Dividing by the estimated background reduces the effects of non-uniform illumination and preserves the contrast between foreground and background, facilitating subsequent segmentation steps.

6.14.5.4 📦 Input and Output Specification (VPL)

Input:

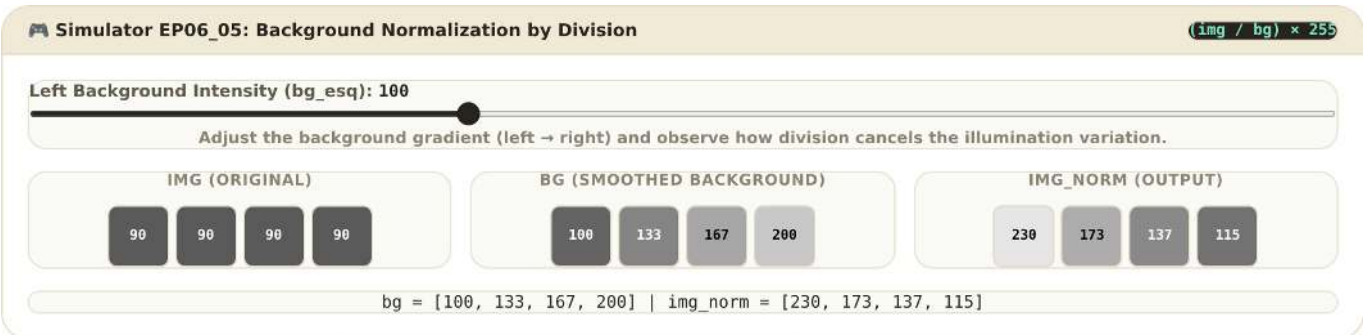
- Line 1: Integer L.
- Line 2: Integer C.
- Next L lines: elements of the matrix `img`.
- Next L lines: elements of the matrix `bg`.

Output:

- Matrix `img_norm`, with L rows and C columns, containing integer values separated by spaces.

6.14.5.5 📌 Examples

Input	Output	Observation
2 2 60 120 180 40 100 100 200 80	153 255 230 128	Values greater than 255 must be saturated; $180/200 \times 255 = 229.5$ results in 230 after rounding.
1 3 30 60 90 60 60 60	128 255 255	Only the first value remains below 255 after normalization.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap06/sim-ep0605.html

Figure 6.25: EP06_05 Simulator: Background Normalization by Division

```
%%writefile EP06_05.py
# Python code
```

```
Overwriting EP06_05.py
```

```
TestSuite("EP06_05.py").run()
```

```
<IPython.core.display.HTML object>
```

6.14.6 EP06_06 🔴 Local Variance Map for Texture Detection

A fabric factory needs to inspect rolls of cloth in real time, without having a reference image available — each roll presents small natural variations. In this situation, the strategy presented in the chapter consists of analyzing the **local homogeneity of the texture**: uniform regions exhibit low intensity variance in small neighborhoods, while scratches, stains, and manufacturing defects produce local increases in this variance.

In this exercise, you will implement the core of this method, calculating the local variance in a sliding window and generating a binary mask that identifies regions whose variance exceeds a threshold.

6.14.6.1 📋 Implementation Guidelines

1. **Dimensions and parameters:** Read the integers L, C, k (window size, always odd) and T (variance threshold).
2. **Image:** Read the $L \times C$ integer values of the texture matrix (intensities between 0 and 255).
3. **Border handling:** When the window extends beyond the image boundaries, use **border replication**, i.e., repeat the value of the nearest valid pixel.
4. **Local mean:** For each position (i, j), compute

$$\mu(i,j) = \frac{1}{k^2} \sum_{(p,q)\in\text{window}} \text{texture}(p,q).$$

5. **Local variance:** Compute the population variance of the window,

$$\sigma^2(i,j) = \frac{1}{k^2} \sum_{(p,q)\in\text{window}} \left(\text{texture}(p,q) - \mu(i,j)\right)^2,$$

   or, equivalently,

$$\sigma^2(i,j) = \overline{x^2} - \mu(i,j)^2,$$

   where $\overline{x^2}$ represents the mean of the squared intensities.
6. **Rounding:** Round the variance to the nearest integer (*round half away from zero*, using `np.floor(res_norm + 0.5)`).
7. **Thresholding:** Set $\text{mask}(i,j) = 1$ if the rounded variance is **strictly greater** than T; otherwise, set $\text{mask}(i,j) = 0$.
8. **Output:** Print the resulting binary mask.

6.14.6.2 📌 Computational Constraints

- **Border replication:** use the value of the nearest valid pixel whenever the window extends beyond the image boundaries.
- **Population variance:** use denominator k^2, never $k^2 - 1$.
- **Strict comparison:** the mask must be computed using the condition $\sigma^2_{\text{rounded}} > T$.
- **Odd window:** the value of k is always odd, ensuring a central pixel.

6.14.6.3 🧠 Theoretical Foundation

Situation	Local variance	Interpretation
Uniform region	Low	Similar intensities in the neighborhood.
Region containing a defect	High	The presence of distinct intensities increases the dispersion of values.
Small window	Greater sensitivity to details and noise	Detects localized changes.
Large window	Smoother response	Highlights larger defects, but reduces the precision of their localization.

Local variance measures the dispersion of intensities in a neighborhood. Homogeneous regions exhibit low variance, while texture changes increase this measure, allowing the identification of potential defects through simple thresholding.

6.14.6.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Integer k (odd).
- Line 4: Integer T.
- Next L lines: integer elements of the texture matrix.

Output:

- Binary mask (values 0 or 1), with L rows and C columns.

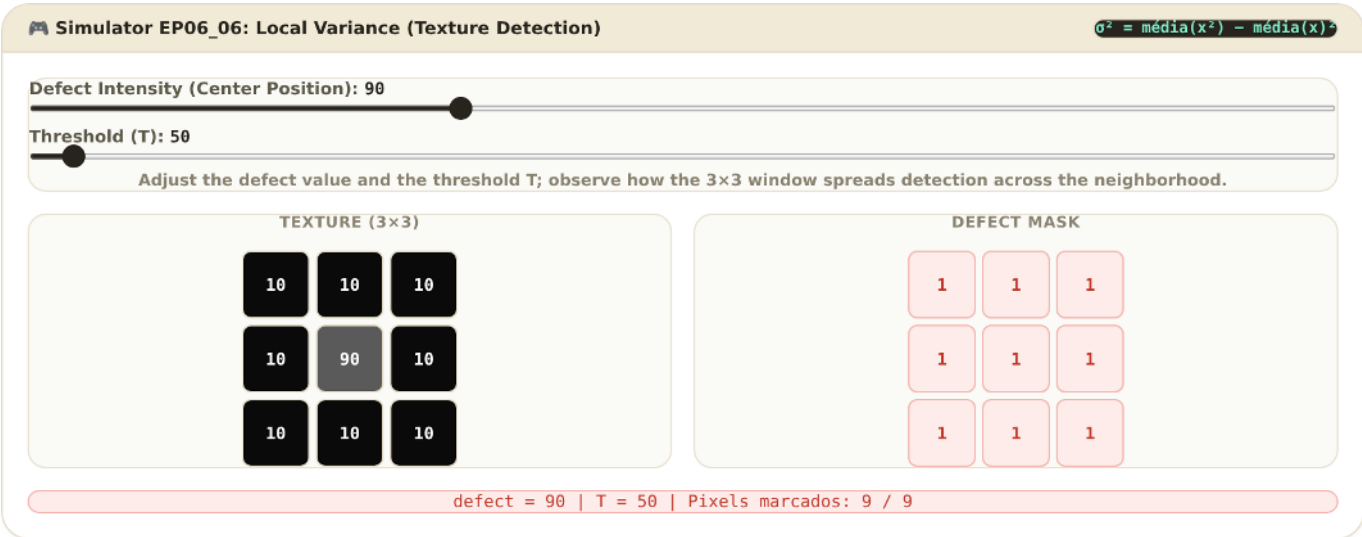
6.14.6.5 📌 Examples

Input	Output	Observation
3 3 3 50 10 10 10 10 10 10 10 90 10	0 0 0 1 1 1 1 1 1	The defect increases the variance in all windows that contain it.
2 2 3 5 100 100 100 100	0 0 0 0	The texture is uniform; the variance is zero throughout the image.

```
%%writefile EP06_06.py
# Python code
```

```
Overwriting EP06_06.py
```

```
TestSuite("EP06_06.py").run()
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap06/sim-ep0606.html

Figure 6.26: EP06_06 Simulator: Local Variance Map for Texture Detection

```
<IPython.core.display.HTML object>
```

6.14.7 EP06_07 🟣 Industrial Inspection Pipeline: Registration by Translation and Subtraction

On a production line, a fixed camera photographs each part as it passes on the conveyor belt, comparing it to a defect-free reference image. The problem: small vibrations in the conveyor shift the part relative to the reference position at each capture. If image subtraction is applied directly, without correction, the displacement alone already generates enormous differences — **false positives** that mask the real defects.

This is the most comprehensive exercise in the chapter: you must **first register** (geometrically align) the captured image using a known displacement (dx, dy), provided by a position sensor on the conveyor, and **only then apply subtraction** with thresholding, exactly as described in the industrial inspection section.

6.14.7.1 📋 Implementation Guidelines

1. **Dimensions and parameters:** Read L, C (image dimensions), the known integer displacement dx, dy (which may be negative), and the detection threshold T (integer).
2. **Images:** Read the reference matrix (`ref`, $L \times C$, defect-free) and the captured matrix (`cap`, $L \times C$, possibly shifted and with defects).
3. **Registration by translation:** Construct the aligned image `alin` by applying the received displacement (dx, dy):
$$\text{alin}(i,j) = \begin{cases} \text{cap}(i+dy,\ j+dx), & \text{if } (i+dy,\ j+dx) \in [0,L) \times [0,C) \\ 0, & \text{otherwise} \end{cases}$$
4. **Border filling:** Positions that "leave" the captured image after the displacement are assigned the value **0** (*zero-padding* — outside the camera's field of view; **note that this exercise uses zero, unlike the border replication of EP06_06**).
5. **Absolute difference:** Compute, pixel by pixel,
$$\text{diff}(i,j) = |\text{ref}(i,j) - \text{alin}(i,j)|$$
6. **Thresholding:** Define $\text{mask}(i,j) = 1$ if $\text{diff}(i,j) > T$; otherwise, $\text{mask}(i,j) = 0$.
7. **Output:** In this order — (a) the matrix `alin` ($L \times C$); (b) the defect mask ($L \times C$); (c) a final line with the total number of pixels classified as defective.

6.14.7.2 📌 Computational Constraints

- ***Zero-padding*, not replication:** positions outside the bounds of the captured image, after the displacement, are exactly 0 — this is the point that most differentiates this exercise from EP06_06.
- **Strict comparison:** $\text{diff}(i, j) > T$.
- **Sign of (dx, dy):** the displacement may be positive or negative; the formula in step 3 must be applied literally, without inverting the signs.
- **All values are integers:** there is no rounding at this stage.

6.14.7.3 🧠 Theoretical Foundation

Omitted step	Consequence
Skipping geometric registration	The entire image border (introduced by the displacement) is marked as "defect" — systematic false positive
Registration with incorrect (dx, dy)	Part and reference remain misaligned; subtraction detects shifted contours, not real defects
Threshold T too low	Capture noise (variations of 1–2 gray levels) is mistaken for defects
Threshold T too high	Subtle defects go undetected

Geometric registration and subtraction are complementary steps: the former ensures that both images represent exactly the same scene in the same spatial reference frame; the latter isolates what actually changed between them — ideally, only the defects.

6.14.7.4 📦 Input and Output Specification (VPL)

Input:

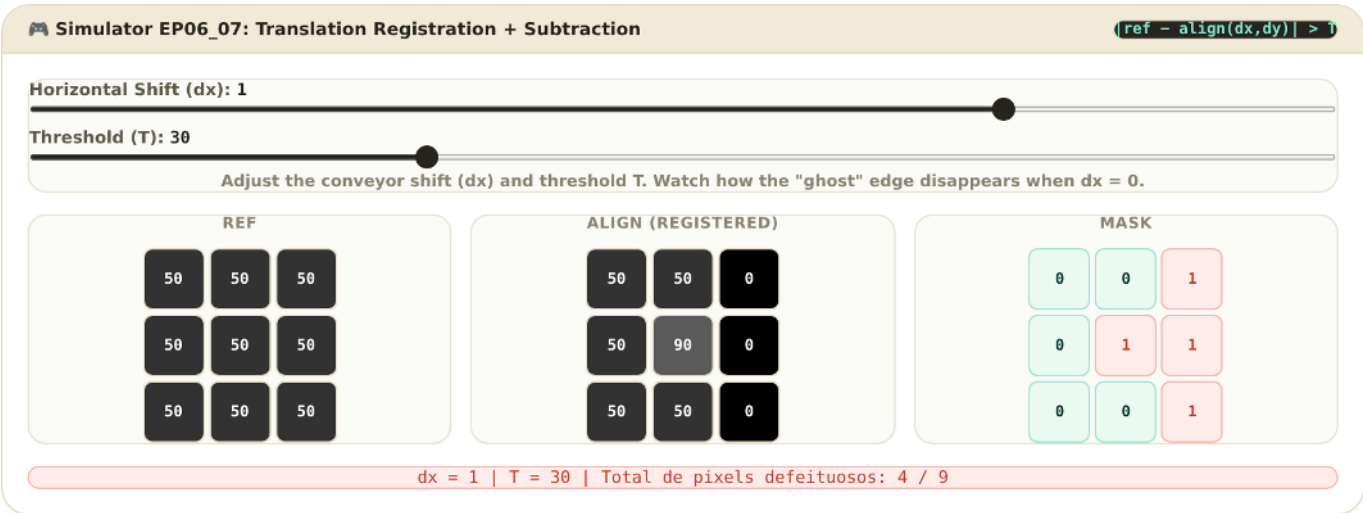
- Line 1: Integer L.
- Line 2: Integer C.
- Line 3: Two integers dx and dy, separated by a space.
- Line 4: Integer T.
- Next L lines: integer elements of the `ref` matrix.
- Next L lines: integer elements of the `cap` matrix.

Output:

- L lines with the `alin` matrix.
- L lines with the defect mask (0/1).
- Last line: `Total de pixels defeituosos: X`.

6.14.7.5 📌 Examples

Input	Output	Remark
3 3 1 0 30 50 50 50 50 50 50 50 50 50 0 50 50 0 50 90 0 50 50	50 50 0 50 90 0 50 50 0 0 0 1 0 1 1 0 0 1 Total de pixels defeituosos: 4	$dx = 1$ shifts the reading one column to the right; the last column of `alin` has no correspondence (becomes 0) and is systematically marked; the real defect (90) is also detected.
2 2 0 0 20 10 10 10 10 10 10 10 60	10 10 10 60 0 0 0 1 Total de pixels defeituosos: 1	No displacement ($dx = dy = 0$): `alin` is identical to `cap`; only the real defect (60) is detected.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap06/sim-ep0607.html

Figure 6.27: EP06_07 Simulator: *Pipeline* Inspection — Registration by Translation and Subtraction

```
%%writefile EP06_07.py
# Python code
```

```
Overwriting EP06_07.py
```

```
TestSuite("EP06_07.py").run()
```

```
<IPython.core.display.HTML object>
```

6.14.8 EP06_08 ⚫ QRCode Segmentation and Real Decoding with OpenCV

In the previous exercises, the intermediate quantities of the image processing pipeline—such as areas, perimeters, variances, and displacements—were provided directly or calculated from numerical matrices, without the need for specialized Computer Vision libraries. In this chapter-closing exercise, this restriction is

intentionally removed: the **OpenCV** library (`cv2`) will be used to locate and decode a real *QRCode* present in a scene.

The proposal reproduces a simplified workflow of systems employed in visual inspection, industrial automation, and automatic document reading. To keep the data input accessible to the educational context, image loading will be integrated into the didactic library `morph`, through the `mm.readImg` function.

The scene is provided in **ASCII PGM (P2)** format and contains a single valid *QRCode*, in addition to several **distractor objects**, such as rectangles, textured noise regions, and isolated blocks. Segmentation based solely on geometric properties—such as area and approximately square shape—is necessary to reduce the search space, but it is not sufficient to identify the correct code. The final confirmation will be performed exclusively by attempting to decode using `cv2.QRCodeDetector`, a procedure compatible with real automatic recognition applications.

6.14.8.1 📋 Implementation Guidelines

1. **Reading dimensions and parameters**

   Read, in this order, from standard input:

   - one line containing the number of rows L;
   - one line containing the number of columns C;
   - one line containing the four algorithm parameters separated by spaces:
     - binarization threshold T (integer);
     - minimum area A_{min} (integer);
     - aspect tolerance tol (real);
     - margin M (integer, in pixels).

2. **Image loading**

   Use the didactic function `f = mm.readImg(L, C)` to read the $L \times C$ values of the grayscale image, obtaining a NumPy array of type `uint8`.

3. **Binarization**

   Apply inverted binary thresholding using the threshold T. Every pixel of the original image with intensity strictly greater than T must be converted to 255, while the remaining ones must assume the value 0.

4. **Contour detection**

   Extract the external connected components using `cv2.findContours(...)` with the parameters:

   - `cv2.RETR_EXTERNAL`;
   - `cv2.CHAIN_APPROX_SIMPLE`.

5. **Geometric filtering**

   For each contour found:

   - compute the bounding rectangle `(x, y, w, h)` using `cv2.boundingRect`;
   - keep only candidates that simultaneously satisfy:

     Minimum area

     $$w \times h > A_{\text{min}}$$

     Aspect ratio

     $$\left|\frac{w}{h} - 1\right| \leq \text{tol}$$

6. **Candidate ordering**

   Sort the candidates by the bounding rectangle area

   $$w \times h$$

   in descending order.

   In case of a tie, preserve the order originally returned by `cv2.findContours`.

7. **Verification by decoding**

   For each candidate, following the established order:

   - expand the rectangle by M pixels in all four directions;
   - clamp the indices to remain within the image;
   - extract the crop directly from the original image `f`;
   - apply `cv2.QRCodeDetector().detectAndDecode(...)` to that crop.

8. **Stopping criterion**

   Immediately stop processing when the first candidate yields a non-empty decoded string.

9. **Case not found**

   If no candidate is successfully decoded, print exactly: `QRCODE_NAO_ENCONTRADO`

10. **Output (found case)**

    Print two lines.

    First line: `linha coluna altura largura` using the **original** bounding rectangle, before the expansion by the margin M.

    Second line: `texto_decodificado`

6.14.8.2 📌 Computational Restrictions

- Use OpenCV functions to perform binarization, contour detection, bounding rectangle computation, and QRCode decoding.
- Geometric filtering must necessarily occur before the decoding step.
- Use exclusively the fixed threshold T provided in the input. It is not allowed to use automatic thresholding methods, such as Otsu or adaptive thresholding.
- Ensure that the crops sent to the decoder remain within the image boundaries.

6.14.8.3 🧠 Theoretical Foundation

Step	Role in the pipeline	Consequence if omitted
Geometric filtering	Reduces the search space by selecting only regions compatible with the expected QRCode geometry.	The decoder would process all contours, including noise and distractor objects.
Decoding	Semantically confirms whether the candidate contains a valid QRCode.	Geometrically similar objects could be incorrectly classified as a QRCode.
Margin M	Preserves the quiet zone around the code, facilitating its detection.	The absence of this margin may prevent proper alignment and correct reading of the code.

This exercise integrates concepts studied throughout the chapter into a single Computer Vision pipeline. Segmentation reduces the set of candidate regions through geometric characteristics, while the decoding step validates the content of the region using a specialized recognition algorithm.

6.14.8.4 📦 Input and Output Specification (VPL)

Input Structure

```
L
C
T A_min tol M
[image matrix]
```

Output Structure (Success)

```
linha coluna altura largura
texto_decodificado
```

Output Structure (Failure)

```
QRCODE_NAO_ENCONTRADO
```

6.14.8.5 📌 Reference Files (.pgm)

For validation purposes, local debugging, and analysis of real pixel matrices, the image files generated in ASCII P2 format are available in the project directory. You may use them to test the adherence of your code by decoding them with your cell phone (save the *.pgm files locally to view them):

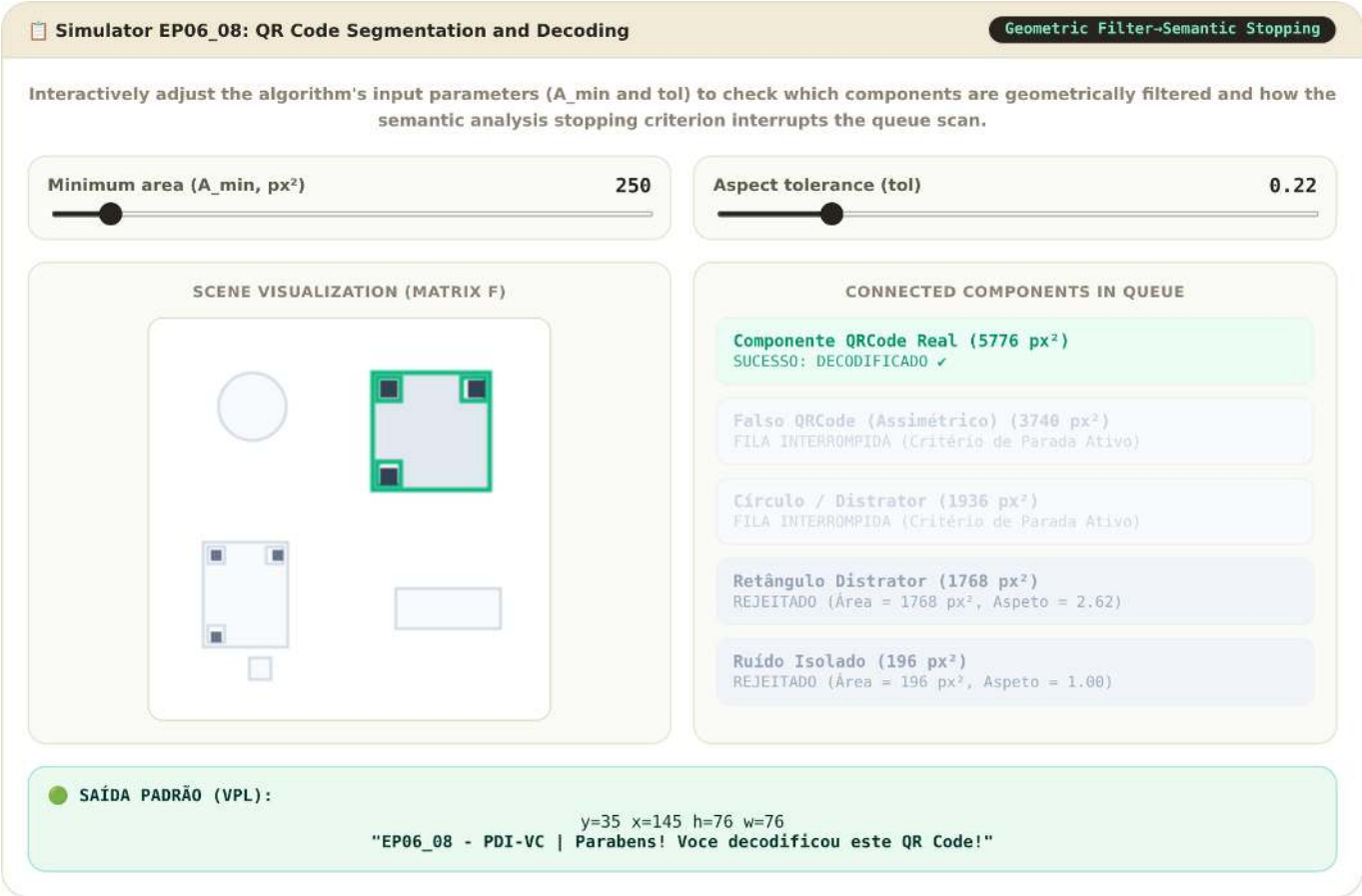
- 📥 **Case 1: Normal Pattern** – Contains a single perfectly centered code with simple geometric distractors on the periphery.
- 📥 **Case 2: Complex Scenario** – Presents a higher density of textured noise and multiple candidate distractors that test the limits of aspect filtering.
- 📥 **Case 3: Expanded Message** – Contains a QRCode structured from a longer character string, generating a higher density of internal modules.
- 📥 **Case 4: Compact Geometry** – Evaluates the pipeline behavior under optimized contrast conditions and borderline positioning.
- 📥 **Case 5: Exclusion Scenario** – An image composed purely of high-area distractor elements, designed to validate the controlled failure behavior of the program.

```
%%writefile EP06_08.py
# Python Code
```

```
Overwriting EP06_08.py
```

```
TestSuite("EP06_08.py").run()
```

```
<IPython.core.display.HTML object>
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap06/sim-ep0608.html

Figure 6.28: Simulador EP06_08: Geometric Segmentation + Verification by QRCode Decoding

Chapter 7

Image Classification and Pattern Recognition

Run Colab Open GitHub

In **Chapter 6**, the transition from Part I to Part II was presented through two applications that already required automated decisions: mark recognition on answer sheets (OMR) and defect detection in industrial inspection. In both cases, however, the decisions relied on geometric rules and manually defined thresholds, such as determining whether a disk was sufficiently circular or whether a region was dark enough.

This chapter formalizes the more general problem underlying these applications: given a set of labeled examples, how can a system be trained to **automatically classify** new images or regions of interest? This question lies at the core of **Pattern Recognition**, a discipline that underpins much of modern Computer Vision tasks, from image classification to object detection and semantic segmentation, explored in the following chapters.

The main **classical image descriptors** (color, texture, and shape/gradient) and the **k-Nearest Neighbors** (*k-NN*) classifier will be studied, chosen for its conceptual simplicity and for directly highlighting the relationship between the feature space, distance metrics, and decision boundaries — concepts that remain central even in classifiers based on deep neural networks, studied in the final chapter of this part.

7.1 Chapter Objectives

By the end of this chapter, the student should be able to:

- **Understand the classical pattern recognition pipeline**: acquisition, preprocessing, descriptor extraction, classification, and evaluation;
- **Extract and interpret classical descriptors** of color, texture (*Local Binary Patterns* — LBP), and shape/gradient (*Histogram of Oriented Gradients* — HOG);
- **Implement and train a k-NN classifier** for image classification tasks;
- **Evaluate classifiers** using metrics such as accuracy, confusion matrix, precision, and recall;
- **Analyze the effect of the k parameter** and the dimensionality of the feature space on classifier performance;
- **Recognize the limitations of hand-crafted features** and understand the motivation for the transition, in the following chapters, to automatically learned descriptors.

7.2 Environment Setup

The examples in this chapter use libraries that are widely employed in Digital Image Processing, Computer Vision, and Machine Learning. The block below installs the necessary packages; in environments where they are already available, the execution can be skipped.

```
import os, urllib.request
```

```python
url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup()
from morph import mm

import importlib
import subprocess
import sys

def setup_cap07():
    """Installs missing libraries needed for this chapter
    (computer vision and machine learning)."""
    pacotes = {
        "cv2": "opencv-python",
        "skimage": "scikit-image",
        "numpy": "numpy",
        "sklearn": "scikit-learn",
        "matplotlib": "matplotlib",
        "pandas": "pandas",
        "seaborn": "seaborn",
        "tabulate": "tabulate",
        "kaleido": "kaleido",
    }
    for modulo, pacote in pacotes.items():
        if importlib.util.find_spec(modulo) is None:
            resultado = subprocess.run(
                [sys.executable, "-m", "pip", "install", "-q", pacote]
            )
            if resultado.returncode != 0:
                print(f"[WARNING] Failed to install {pacote} (needed for the module
                {modulo}).")


setup_cap07()

# ==========================================================
# Libraries
# ==========================================================

# Scientific computing
import numpy as np
import pandas as pd

# Visualization
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import seaborn as sns

# Computer vision
import cv2
from skimage import data as skdata
from skimage.feature import hog, local_binary_pattern

# Machine learning
from sklearn.datasets import load_digits, make_classification
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    accuracy_score,
    classification_report,
    confusion_matrix,
    f1_score,
    precision_score,
    recall_score,
)
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0
```

7.3 A Concrete Problem: Fruit Classification

Before presenting the theoretical foundations, consider the following problem, which will be used as an example throughout this chapter to illustrate the main concepts of pattern recognition.

The scenario: An automated farm uses a Computer Vision system to separate apples, bananas, and oranges on packaging lines.

The challenge: The fruits arrive on the conveyor belt in different positions and orientations, under lighting conditions that may vary. Furthermore, leaves, shadows, and small occlusions can hinder their identification. How can we develop a system capable of classifying them correctly?

A possible approach:

1. Extract descriptors that represent relevant characteristics of the fruits:
   - **Color:** predominant color distribution;
   - **Texture:** differences in the peel surface;
   - **Shape:** geometric characteristics of the contour.
2. Train a classifier using previously labeled examples.
3. Use the trained model to automatically classify new fruits.

Figure 7.1 illustrates, conceptually, how different fruits can be represented in a three-dimensional feature space.

> Reflect before continuing
>
> If each fruit were represented only by the intensity values of its pixels, would it be possible to distinguish them reliably? What types of information could be extracted from the image to facilitate this task?

```
np.random.seed(42)

centros = {
    "Maçã":    [0.8, 0.2, 0.9],
    "Banana":  [0.3, 0.1, 0.2],
    "Laranja": [0.9, 0.8, 0.8],
}

fig = plt.figure(figsize=(5, 5))
ax = fig.add_subplot(projection="3d")

for fruta, centro, cor, marcador in zip(
    centros,
    centros.values(),
    ["red", "gold", "orange"],
```

```
    ["o", "s", "^"],
):
    X = np.clip(np.random.normal(centro, 0.08, (70, 3)), 0, 1)
    ax.scatter(X[:, 0], X[:, 1], X[:, 2],
               c=cor, marker=marcador, s=35,
               alpha=0.7, label=fruta)

ax.set(
    xlim=(0,1), ylim=(0,1), zlim=(0,1),
    xlabel="Intensidade de cor",
    ylabel="Textura",
    zlabel="Forma",
    title="Espaço de Características"
)
ax.view_init(elev=25, azim=-60)
ax.legend(title="Frutas")
ax.zaxis.labelpad = 0.01

plt.tight_layout()
plt.show()
```

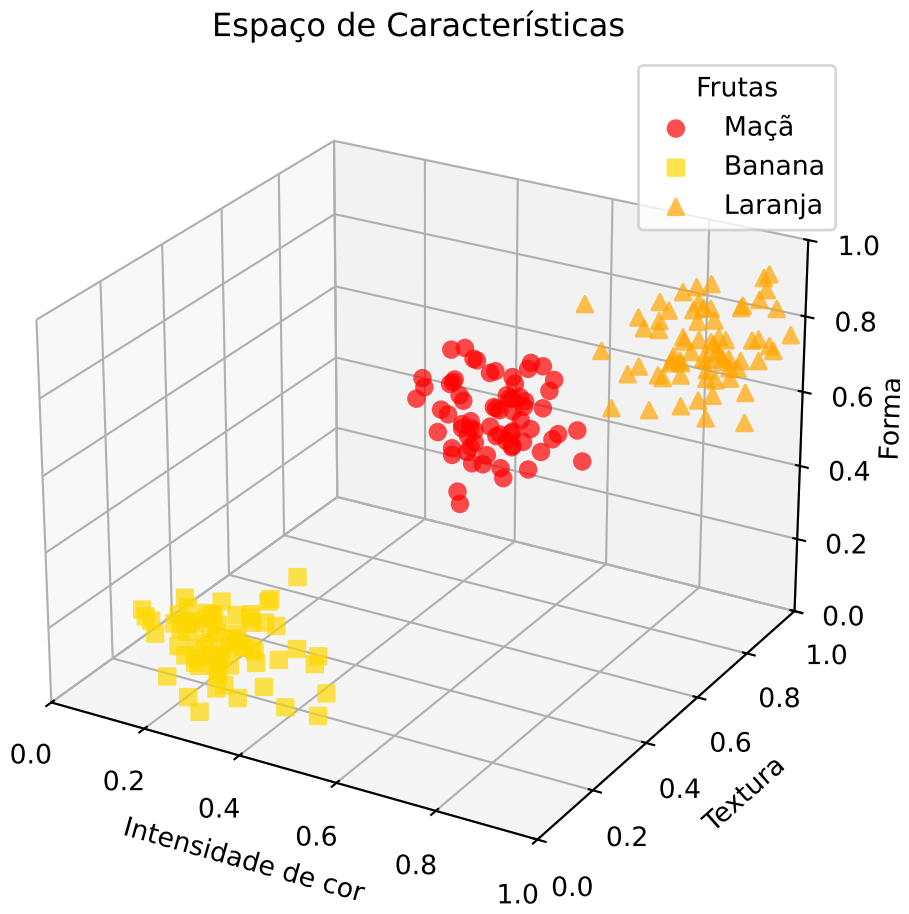


Figure 7.1: Motivational example: different fruits forming distinct clusters in a feature space.

7.4 Chapter Overview: The Classic Image Classification *Pipeline*

Before proceeding, it is useful to present an integrated view of what will be studied. Figure 8.1 shows the general flow of a classic image classification system, from the input image to the final label assignment stage. In the following sections, each of the stages of this process will be studied in detail.

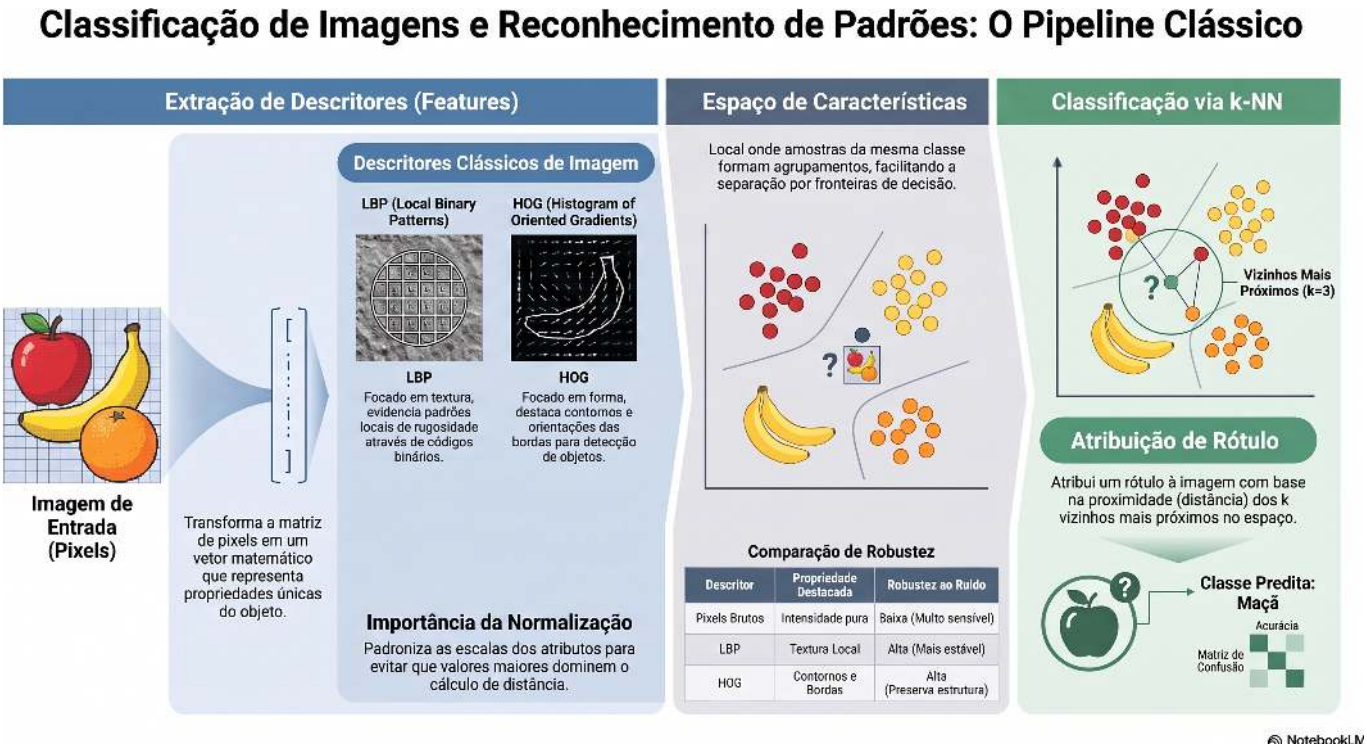


Figure 7.2: Overview of the classic image classification *pipeline*: descriptor extraction (LBP, HOG), formation of the feature space, and classification via k-NN. Not applicable to Deep Learning models (CNNs, YOLO), which learn *features* end-to-end directly from pixels. **Source:** prepared with the aid of Gemini Notebook ({GOOGLE}, 2025).

> **ℹ Scope of this chapter**
>
> In this chapter, the focus falls exclusively on the k-NN classifier, due to its didactic simplicity and its intuitive illustration of the concept of feature space. Other traditional classifiers widely used in Pattern Recognition — such as Decision Trees, Classification Rules, and Support Vector Machines (SVM) — are discussed in depth in Quilici-gonzalez (2014), especially in its 2nd edition, currently in production (QUILICI-GONZALEZ, 2026).

7.5 Fundamentals of Pattern Recognition

A **pattern recognition** system aims to assign a category (label) to an observation — an entire image, a region of interest, or a signal — based on previously labeled examples. In general, this process is organized into the following steps:

1. **Acquisition:** obtaining the image or signal to be classified;
2. **Pre-processing:** normalization, noise removal, geometric or illumination correction — steps already studied in previous chapters;
3. **Feature extraction:** transformation of the image into a **feature vector** of fixed dimension, which represents the relevant properties for the classification task;
4. **Classification:** application of a model that associates the feature vector with a class;
5. **Evaluation:** analysis of the model's performance on a dataset independent of the one used for training.

The set of all possible feature vectors constitutes the **feature space**. A good descriptor produces representations that, in this space, bring observations of the same class closer together and separate observations of distinct classes. This property favors classification methods based on proximity, such as **k-NN**, and also benefits several other classifiers.

Figure 7.1 illustrates this concept schematically: each fruit is represented by a point in a three-dimensional feature space (color, texture, and shape). Although this space is merely a didactic simplification, it shows how samples from the same class tend to form clusters, while different classes occupy distinct regions, facilitating the classification task.

7.6 Extraction of Classical Descriptors

Before the popularization of deep neural networks, image descriptors were mostly hand-crafted by specialists, based on known statistical or geometric properties. Three classical families are particularly relevant:

- **Color descriptors:** intensity or hue histograms, which capture the distribution of color values in a region, already introduced in **Chapter 3** through the `mm.hist` function;
- **Texture descriptors:** capture local patterns of repetition, roughness, or orientation, such as *Local Binary Patterns* (LBP), studied below;
- **Shape/gradient descriptors:** describe the distribution of edges and gradient orientations, such as the *Histogram of Oriented Gradients* (HOG), widely used in the detection of people and other objects.

To compare the information captured by each approach, Figure 7.3 shows how different techniques "see" the same image.

```
# Load example image
imagem = skdata.camera()

# Apply descriptors
lbp_img = local_binary_pattern(
    imagem,
    P=8,
    R=1,
    method="uniform"
    )

# Convert LBP to RGB only to facilitate visualization
lbp_norm = (lbp_img - lbp_img.min()) / (lbp_img.max() - lbp_img.min() + 1e-8)
lbp_rgb = (cm.nipy_spectral(lbp_norm)[..., :3] * 255).astype("uint8")

hog_features, hog_img = hog(
    imagem,
    orientations=9,
    pixels_per_cell=(8, 8),
    cells_per_block=(2, 2),
    visualize=True,
)

# Standardized display
mm.show(
    [imagem, lbp_rgb, hog_img],
    titles=[
        "Original Image\n(as humans see it)",
        "LBP: Texture\n(each color = an LBP code)",
        "HOG: Gradients and Edges\n(bright regions = higher intensity)",
    ],
    cols=3,
    figsize=(12, 4),
)

print("Notice how each descriptor highlights different properties:")
print("• LBP: highlights local texture patterns.")
print("• HOG: highlights edges and border orientations.")
print("• Original image: contains only intensity values.")
```

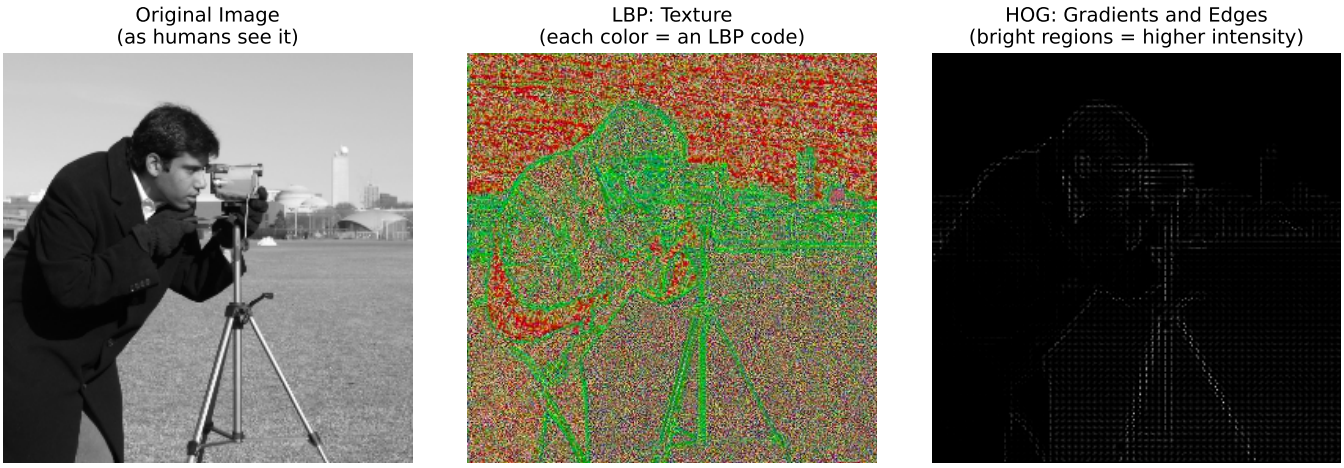


Figure 7.3: Visual comparison of different descriptors applied to the same image. Each descriptor reveals distinct aspects of the scene.

```
Notice how each descriptor highlights different properties:
• LBP: highlights local texture patterns.
• HOG: highlights edges and border orientations.
• Original image: contains only intensity values.
```

7.6.1 *Local Binary Patterns* (LBP)

LBP is a texture descriptor that encodes, for each central pixel g_c, the relationship between its intensity and that of the P neighbors arranged in a circular neighborhood of radius R:

$$\mathrm{LBP}_{P,R}(x_c, y_c) = \sum_{p=0}^{P-1} s(g_p - g_c)\, 2^p, \qquad s(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

where:

- (x_c, y_c) are the coordinates of the central pixel;
- g_c is the intensity of the central pixel;
- g_p is the intensity of the p-th neighboring pixel;
- P is the number of neighbors considered;
- R is the radius of the circular neighborhood;
- p is the neighbor index, with $p = 0, 1, \ldots, P-1$;
- $s(z)$ is the threshold function defined in the equation, where $z = g_p - g_c$; it assumes a value of 1 when $z \geq 0$ and 0 when $z < 0$;
- 2^p corresponds to the binary weight associated with the p-th neighbor.

The resulting LBP code describes the local contrast pattern around the pixel. The histogram of these codes forms a compact feature vector for representing image texture (Figure 7.4). In this chapter, the **uniform** variant is used, which groups non-uniform patterns into a single category, reducing dimensionality and increasing the robustness of the descriptor.

```
plt.figure(figsize=(6, 4))

plt.hist(
    lbp_img.ravel(),
    bins=np.arange(-0.5, lbp_img.max() + 1.5, 1),
    density=True,
    edgecolor="black",
)
```

```
plt.title("Histograma dos códigos LBP")
plt.xlabel("Código LBP")
plt.ylabel("Frequência relativa")
plt.xticks(range(int(lbp_img.max()) + 1))
plt.grid(axis="y", alpha=0.3)

plt.tight_layout()
plt.show()
```

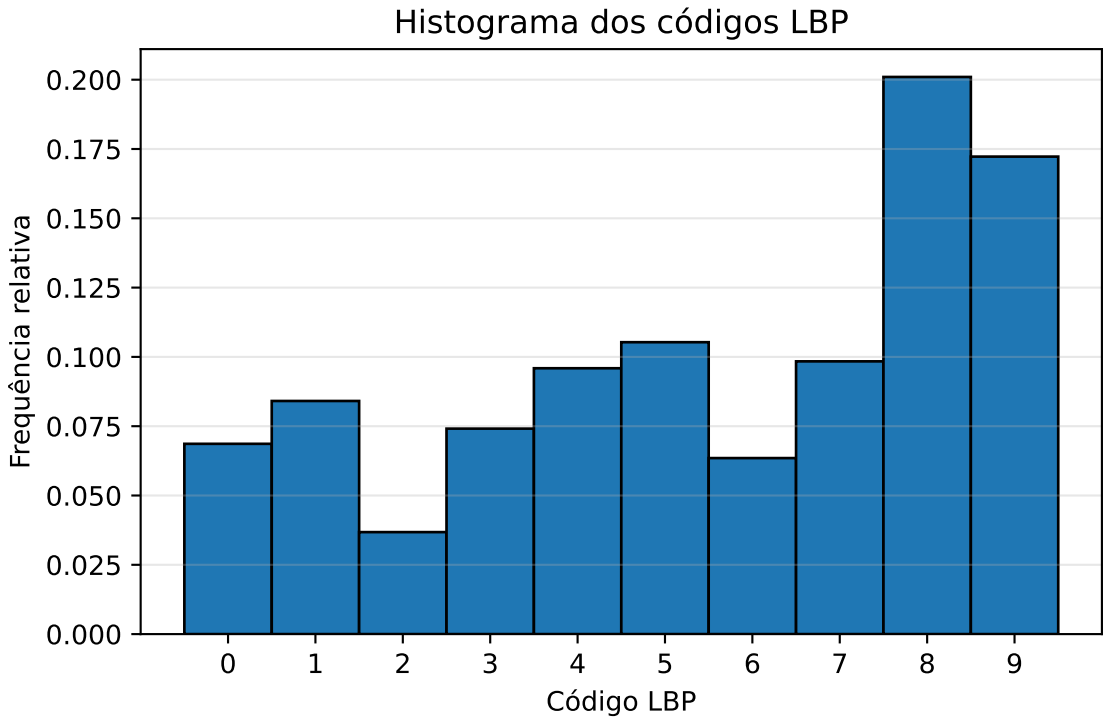


Figure 7.4: Histogram of LBP codes from the cameramen image. Each bar represents the relative frequency of an LBP code, forming the feature vector used to describe the texture.

Function `local_binary_pattern`

The implementation used in this chapter is provided by the library `scikit-image`:

```
local_binary_pattern(
    imagem,
    P=8,
    R=1,
    method="uniform"
)
```

where:

- `image`: grayscale image;
- `P`: number of equally spaced neighbors in the circular neighborhood;
- `R`: radius of the neighborhood, in pixels;
- `method`: encoding strategy. In this chapter, the value `"uniform"` is used.

The equation presented earlier describes the **original LBP**. In the implementation adopted in this chapter, the `method="uniform"` option initially calculates this code and then remaps non-uniform patterns into a single category, reducing the dimensionality of the descriptor and making it more robust to small local variations.

Figure 7.3 presents the visual representation of the LBP, while Figure 7.4 shows the histogram of LBP codes used as a feature vector.

Practical Project 2 (section **Comparison of Descriptors for Texture Classification**) employs LBP in the classification of different types of synthetic texture.

7.6.2 *Histogram of Oriented Gradients* (HOG)

HOG is a descriptor that represents the shape of an object through the distribution of local gradient orientations. As in the Canny operator (**Chapter 6**), the gradient is initially computed:

$$|\nabla f(x,y)| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}, \qquad \theta(x,y) = \operatorname{atan2}\left(\frac{\partial f}{\partial y}, \frac{\partial f}{\partial x}\right).$$

where:

- $f(x,y)$ is the image intensity at pixel (x,y);
- $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ are, respectively, the partial derivatives of the image in the horizontal and vertical directions;
- $|\nabla f(x,y)|$ is the magnitude of the gradient vector at pixel (x,y), indicating the strength of the local variation in the image;
- $\theta(x,y)$ is the orientation of the gradient vector at pixel (x,y), computed by the function atan2, whose result belongs to the interval $(-\pi,\pi]$.

Although $\theta(x,y)$, as computed by the function atan2, belongs to the interval $(-\pi,\pi]$, the standard HOG implementation uses the **unsigned gradient**: opposite orientations (e.g., 0 and π) are treated as equivalent, and the angles are mapped to the interval $[0,\pi)$ before constructing the histogram. This choice makes the descriptor invariant to the direction of contrast (e.g., a light-dark edge and a dark-light edge produce the same orientation).

The image is then divided into **cells**. For each cell, a histogram of gradient orientations is constructed, weighted by the corresponding magnitude. The concatenation of the histograms from all cells forms the HOG feature vector, which represents the spatial distribution of gradient orientations and captures information about the shape and contours of the object (Figure 7.5).

```
n = 100

plt.figure(figsize=(8, 3))
plt.bar(
    range(n),
    hog_features[:n],
    width=0.9
)

plt.title("Primeiros componentes do vetor HOG")
plt.xlabel(f"Índice do componente (0-{n-1}, de um total de {hog_features.shape[0]})")
plt.ylabel("Valor normalizado")
plt.grid(axis="y", alpha=0.3)

plt.tight_layout()
plt.show()
```

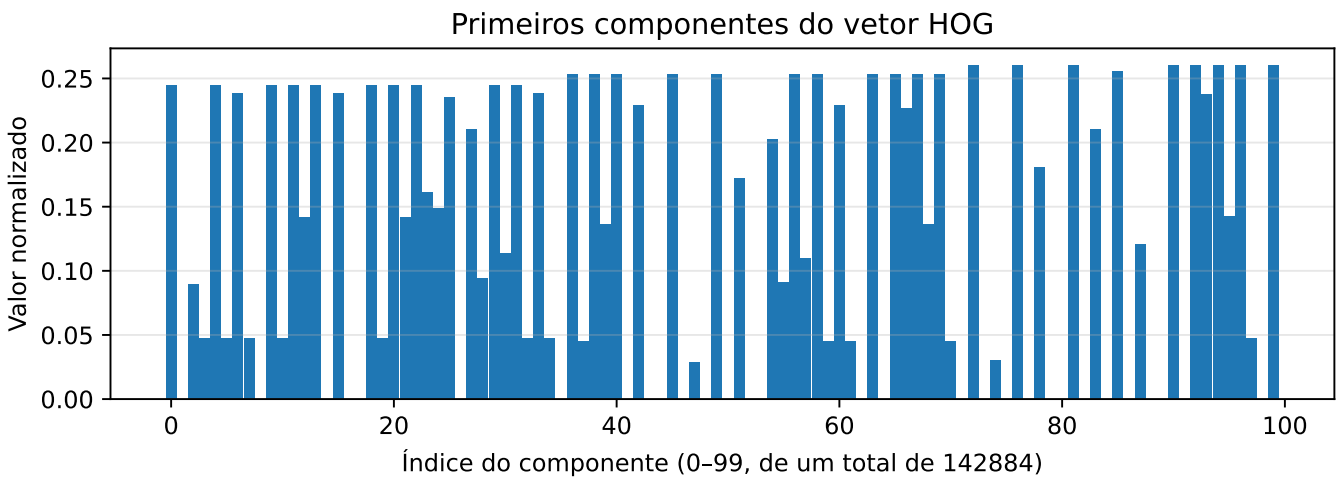


Figure 7.5: First 100 components of the HOG feature vector.

Function `hog`

The extraction of the HOG descriptor is performed by the function:

```
hog(
    image,
    orientations=9,
    pixels_per_cell=(8, 8),
    cells_per_block=(2, 2),
    visualize=True,
)
```

The main parameters are:

- `image`: input image;
- `orientations`: number of angular divisions of the orientation histogram in each cell;
- `pixels_per_cell`: size, in pixels, of each cell where the histogram is computed;
- `cells_per_block`: number of cells used in descriptor normalization;
- `visualize`: when `True`, it also returns an image illustrating the gradients used by the HOG.

The equation presented previously describes the computation of the gradient magnitude and orientation, which form the basis of the HOG descriptor. In the implementation adopted in this chapter, the `hog()` function uses this information to build orientation histograms in each cell of the image and then performs block normalization (`cells_per_block`), reducing the descriptor's sensitivity to variations in illumination and contrast.

Figure 7.3 presents the visual representation of the HOG, while Figure 7.5 illustrates the first components of the feature vector extracted from the image.

Practical Project 1 (section **Classification of Handwritten Digits with k-NN**) compares the performance of HOG descriptors with the direct use of pixel intensities as the feature vector.

7.6.3 The Impact of Scale and Feature Normalization

The k-NN classifier makes its decisions based on the distance between feature vectors. Therefore, the scale of each feature directly influences the classification outcome. If a variable has values much larger than the others (for example, a color intensity ranging from 0 to 255, while a circularity index ranges from 0 to 1), it tends to dominate the distance calculation, reducing the influence of the other descriptors.

To avoid this issue, a **feature normalization** step is applied, typically through standardization (*Z-score standardization*). In this procedure, each feature is adjusted to have a mean of zero and a standard deviation

of one, making quantities originally measured on different scales comparable.

Standardization is performed through the transformation

$$z = \frac{x - \mu}{\sigma},$$

where:

- x is the original value of the feature;
- μ is the mean of that feature computed over the training set;
- σ is the standard deviation of the feature;
- z is the standardized value.

After this transformation, all features have a mean equal to zero and a standard deviation equal to one, allowing them to contribute in a balanced manner to the distance calculation.

`StandardScaler` Class

The standardization used in this chapter is performed by the `StandardScaler` class from the `scikit-learn` library:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_norm = scaler.fit_transform(X)
```

where:

- `StandardScaler()`: creates the object responsible for standardization;
- `fit_transform(X)`: computes the mean and standard deviation of each feature in the set `X` and returns the standardized matrix.

In practice, the `fit_transform()` method performs two steps: first (`fit`), it estimates the mean (μ) and the standard deviation (σ) of each feature; then (`transform`), it applies the standardization transformation presented earlier to all values in the input matrix.

Figure 7.6 shows the effect of normalization. Visually, the distribution of the points remains the same; what changes is the scale of the axes. Without normalization, the feature with the greatest magnitude dominates the distance calculation between samples. After standardization, all features contribute in a balanced manner to the distance calculation used by the k-NN classifier.

```
# Synthetic data with very different scales
np.random.seed(42)
X_demo = np.random.randn(20, 2) * [100, 1]
y_demo = np.array([0] * 10 + [1] * 10)

print("Effect of normalization:")
print("  Feature 1: scale   100")
print("  Feature 2: scale   1")
print("\nWithout normalization, the first feature dominates the distance calculation.")
print("With normalization, both contribute in a balanced way.")
print("\nNormalization is essential when features have different scales.")
fig, axes = plt.subplots(1, 2, figsize=(10, 4))

# Without normalization
axes[0].scatter(
    X_demo[y_demo == 0, 0], X_demo[y_demo == 0, 1],
    c="blue", label="Classe 0"
)
axes[0].scatter(
```

```
    X_demo[y_demo == 1, 0], X_demo[y_demo == 1, 1],
    c="red", label="Classe 1"
)
axes[0].set_title("Sem Normalização\n(escalas diferentes)")
axes[0].set_xlabel("Característica 1 (escala 100)")
axes[0].set_ylabel("Característica 2 (escala 1)")
axes[0].legend()

# With normalization
scaler = StandardScaler()
X_norm = scaler.fit_transform(X_demo)

axes[1].scatter(
    X_norm[y_demo == 0, 0], X_norm[y_demo == 0, 1],
    c="blue", label="Classe 0"
)
axes[1].scatter(
    X_norm[y_demo == 1, 0], X_norm[y_demo == 1, 1],
    c="red", label="Classe 1"
)
axes[1].set_title("Com Normalização\n(características balanceadas)")
axes[1].set_xlabel("Característica 1")
axes[1].set_ylabel("Característica 2")
axes[1].legend()

plt.tight_layout()
plt.show()
```

```
Effect of normalization:
  Feature 1: scale   100
  Feature 2: scale   1

Without normalization, the first feature dominates the distance calculation.
With normalization, both contribute in a balanced way.

Normalization is essential when features have different scales.
```

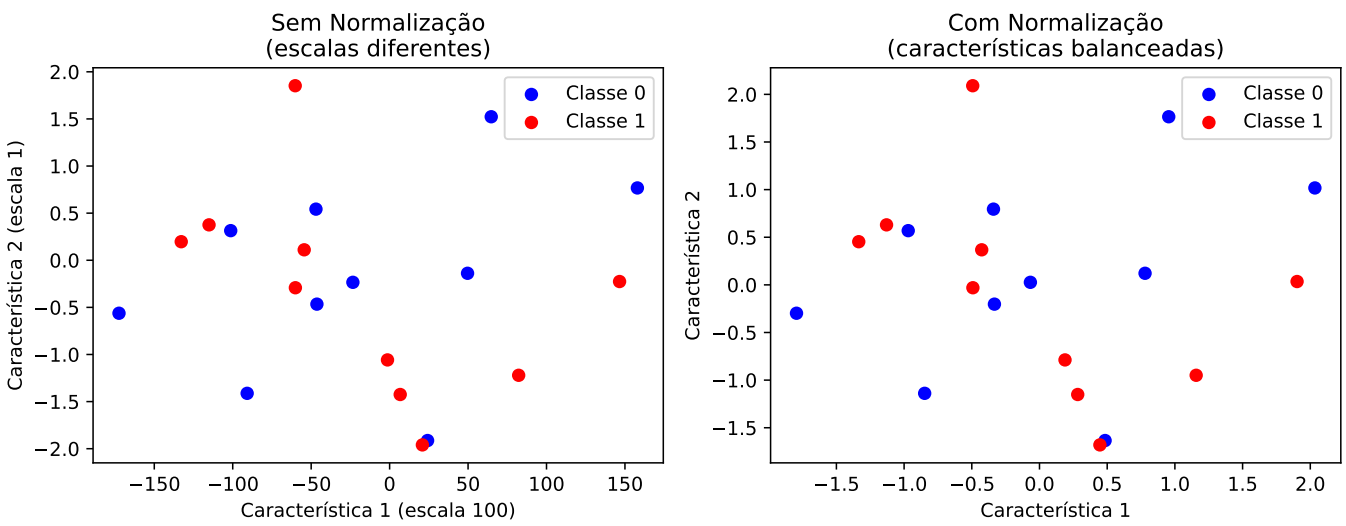


Figure 7.6: Importance of feature normalization for the k-NN classifier.

7.7 📌 Concept Map

Up to this point, we have presented the classical descriptors (LBP, HOG, raw pixels) and how they organize samples in a feature space. Figure 7.7 synthesizes this path and situates these steps within the overall flow of a classical image classification system, also indicating the subsequent steps — classification (k-NN) and result evaluation — which will be formalized in the following sections.

7.8 Descriptors in Practice

After learning about the main classical descriptors, it is natural to ask how they influence the performance of a classifier in situations close to those encountered in practice.

In this section, we compare the use of three distinct representations of the same images: pixel intensities, LBP descriptors, and HOG descriptors. To make the experiment more realistic, synthetic noise is added to the data, at different intensities for each descriptor — a simplified way of simulating the fact that, in practice, different representations tolerate capture imperfections unequally (sensor noise, small position variations, etc.).

Figure 7.8 presents the confusion matrices obtained for each descriptor, allowing us to identify in which classes the main classification errors occur. The interpretation of these matrices was introduced in **Chapter 1**, when the concepts of **True Positive (TP)**, **False Positive (FP)**, **True Negative (TN)**, and **False Negative (FN)** were presented. These concepts were explored in **EPs 01_02** (classification metrics) and **01_03** (*mean Average Precision* – mAP), available at:

- https://fzampirolli.github.io/pdi-vc/eps/py.pt/EP01_02.html
- https://fzampirolli.github.io/pdi-vc/eps/py.pt/EP01_03.html

In this chapter, confusion matrices are used to analyze how different descriptors influence classifier performance.

Next, Figure 7.9 summarizes the overall accuracy obtained by each descriptor.

The results show that classifier performance depends directly on the representation chosen to describe the images. While the direct use of pixel intensities is more sensitive to the introduced degradations, LBP and HOG descriptors better preserve the information relevant to classification, resulting in higher performance in this scenario. It is important to emphasize that the noise levels applied to each descriptor were chosen for didactic purposes only, so as to illustrate the general principle that more elaborate descriptors *may* be more robust to degradations — which does not mean that this relationship always holds, as the case study in the next section will demonstrate.

How the experiment is conducted

Since the goal of this section is to compare only the effect of the descriptors, a simple synthetic dataset is generated: three Gaussian point clouds, centered at the same "color, texture, and shape" values already used in Figure 7.1 — the same pattern employed since the beginning of the chapter to represent the three fruit classes.

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
```

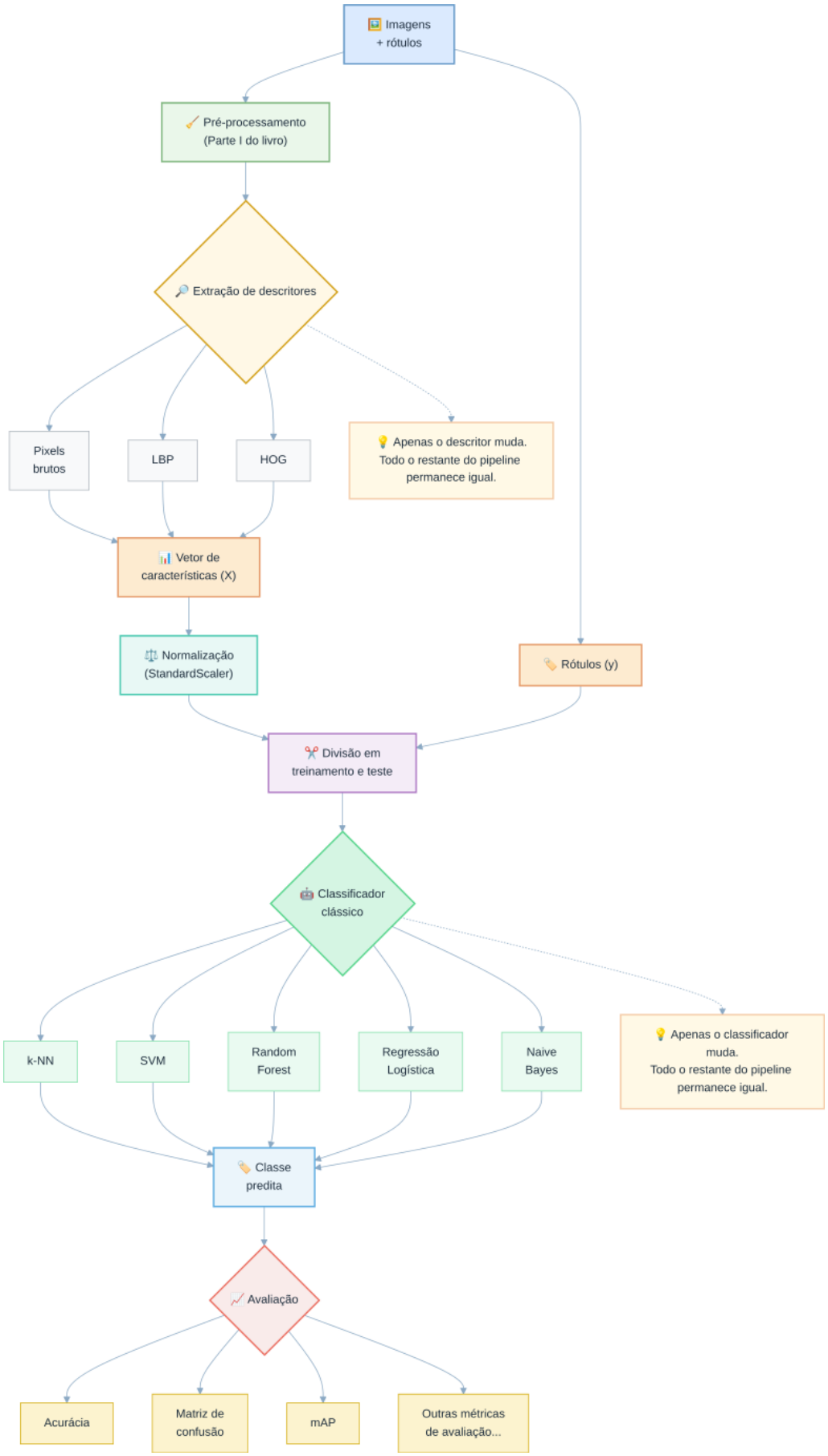


Figure 7.7: Concept map of the image classification process using classical descriptors (LBP, HOG, raw pixels) and a traditional classifier (k-NN). It does not apply to Deep Learning models (CNNs, YOLO), which learn *features end-to-end* directly from pixels.

```
centros = {
    "Maçã":    [0.8, 0.2, 0.9],
    "Banana":  [0.3, 0.1, 0.2],
    "Laranja": [0.9, 0.8, 0.8],
}

n_por_classe = 100
X = np.vstack([
    np.random.normal(centro, 0.12, (n_por_classe, 3))
    for centro in centros.values()
])
y = np.repeat(list(centros.keys()), n_por_classe)
```

where:

- `centros`: dictionary with the mean point of each class in the feature space (color, texture, shape);
- `n_por_classe`: number of samples generated per class;
- `np.random.normal(centro, 0.12, (n_por_classe, 3))`: generates `n_por_classe` samples around each center, with a standard deviation of 0.12 in each dimension;
- `np.repeat(list(centros.keys()), n_por_classe)`: generates the corresponding label vector, in the same order as the centers.

Next, the training and evaluation pipeline is used:

- `train_test_split(X, y, test_size=0.3)`: splits the data into training (70%) and testing (30%);
- `KNeighborsClassifier(n_neighbors=5)`: creates a k-NN classifier with $k = 5$ neighbors;
- `fit(X_train, y_train)`: fits the model to the training data;
- `predict(X_test)`: classifies the test samples;
- `accuracy_score(y_test, y_pred)`: computes the accuracy;
- `confusion_matrix(y_test, y_pred)`: generates the confusion matrix.

In this experiment, the training set, the classifier, and the evaluation method remain exactly the same. The only difference between the experiments is the representation used for each image (raw pixels, LBP, or HOG), allowing the exclusive evaluation of the influence of the descriptor on the classifier's performance.

```
classes = ["Maçã", "Banana", "Laranja"]

np.random.seed(42)

# Same class centers (color, texture, shape) used in the
# previous example, now reused to generate the synthetic
# training and testing data for this experiment.
centros = {
    "Maçã":    [0.8, 0.2, 0.9],
    "Banana":  [0.3, 0.1, 0.2],
    "Laranja": [0.9, 0.8, 0.8],
}

n_por_classe = 100
X = np.vstack([
    np.random.normal(centro, 0.12, (n_por_classe, 3))
    for centro in centros.values()
])
y = np.repeat(list(centros.keys()), n_por_classe)

# Simulate descriptors with different levels of sensitivity to noise.
# The more noise added, the worse the representation tends to be.
descritores = {
    "Pixels Brutos": X + 0.5 * np.random.randn(*X.shape),
```

```
    "LBP":            X + 0.3 * np.random.randn(*X.shape),
    "HOG":            X + 0.2 * np.random.randn(*X.shape),
}

# Create a single figure with 3 subplots side by side for the matrices
fig, axes = plt.subplots(1, 3, figsize=(14, 4))
resultados = {}

for idx, (nome, Xd) in enumerate(descritores.items()):
    X_train, X_test, y_train, y_test = train_test_split(Xd, y, test_size=0.3, random_state=42)

    knn = KNeighborsClassifier(n_neighbors=5)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)

    acc = accuracy_score(y_test, y_pred)
    resultados[nome] = acc

    # Confusion matrix in the corresponding subplot

    cm = confusion_matrix(y_test, y_pred, labels=classes)

    sns.heatmap(
        cm,
        annot=True,
        fmt="d",
        cmap="Blues",
        cbar=False,
        xticklabels=classes,
        yticklabels=classes,
        ax=axes[idx],
    )

    axes[idx].set_title(
        f"{nome}\nAcurácia: {acc:.3f}",
        fontsize=11,
        fontweight="bold"
    )
    axes[idx].set_xlabel("Classe Predita")
    axes[idx].set_ylabel("Classe Real")


plt.tight_layout()
plt.show()
```

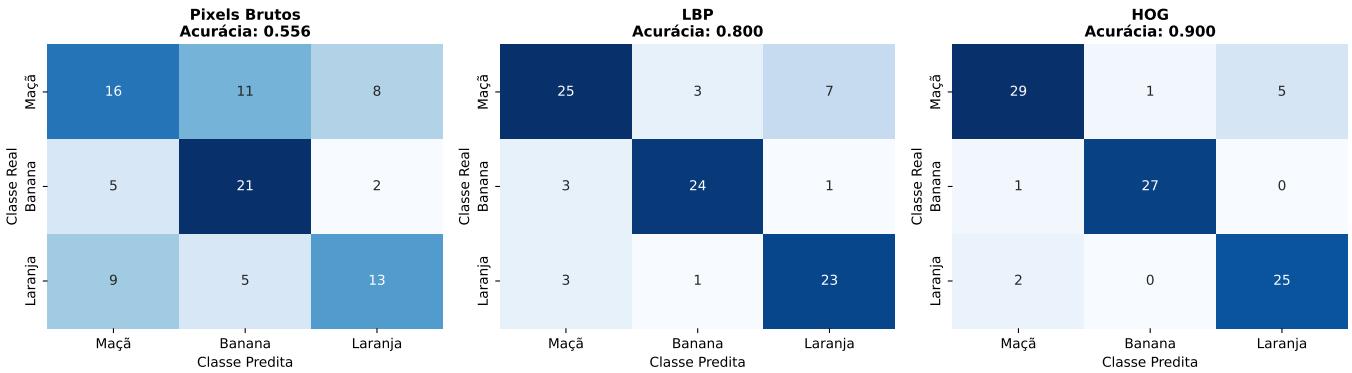


Figure 7.8: Confusion matrices obtained by the k-NN classifier using three different descriptors. The rows represent the actual class (Apple, Banana, and Orange) and the columns the predicted class. The greater the concentration of values on the main diagonal, the better the descriptor's performance.

```
# Visual comparison in a standalone figure
plt.figure(figsize=(6, 3.5))
nomes = list(resultados.keys())
acuracia = list(resultados.values())
colors = ['#6366f1', '#f97316', '#22c55e']

bars = plt.bar(nomes, acuracia, color=colors, width=0.5)
plt.ylabel('Acurácia Global')
plt.title('Desempenho Geral dos Descritores sob Ruído Realista', fontsize=12,
fontweight='bold')
plt.ylim(0.5, 1.0)
plt.grid(axis='y', linestyle='--', alpha=0.5)

# Add the values above the bars using standard rounding for display
for bar, val in zip(bars, acuracia):
    plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
             f'{val:.3f}', ha='center', fontweight='bold', fontsize=10)

plt.tight_layout()
plt.show()

print("Analysis of results:")
print("- Raw Pixels: Sensitive to local lighting variations and noise.")
print("- LBP: Good tolerance to global monotonic lighting variations.")
print("- HOG: Excellent for contours and stable shapes under small geometric fluctuations.")
```

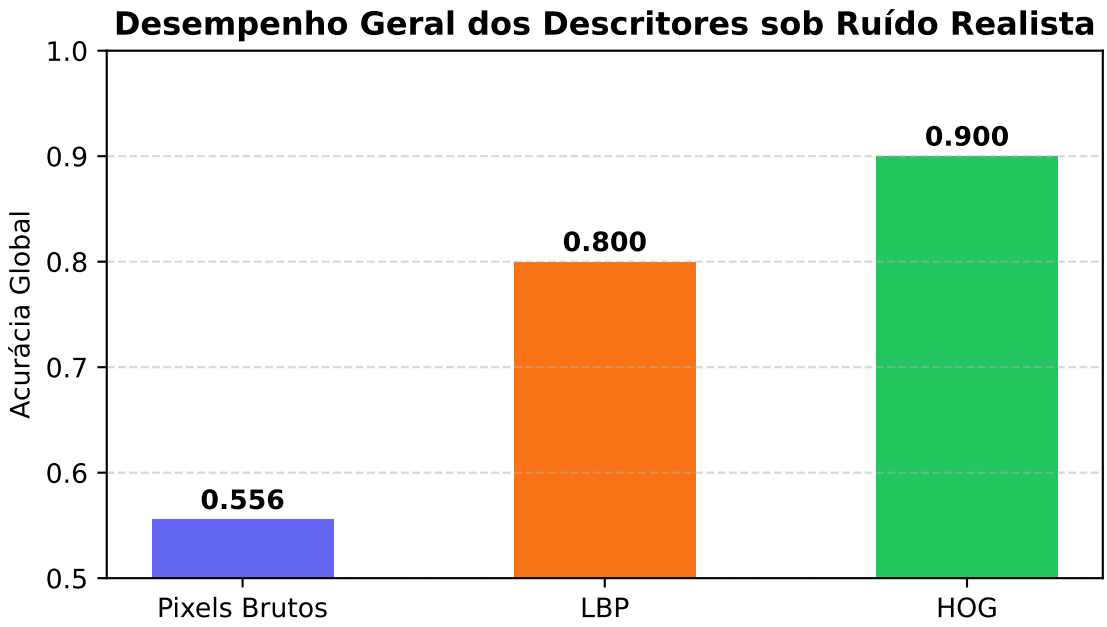


Figure 7.9: Detailed comparison of global accuracy in a realistic scenario. Note how structurally extracted descriptors (LBP and HOG) outperform the use of pure raw pixel intensities.

```
Analysis of results:
- Raw Pixels: Sensitive to local lighting variations and noise.
- LBP: Good tolerance to global monotonic lighting variations.
- HOG: Excellent for contours and stable shapes under small geometric fluctuations.
```

7.9 A Concrete Problem: Simulating Fruit Descriptors

Returning to the fruit classification problem presented at the beginning of the chapter, each image can be represented by a feature vector obtained from the extraction of color, texture, and shape descriptors. Table 7.1 presents some descriptors frequently used in Computer Vision applications, including techniques introduced in Chapter 3 and in this chapter.

Table 7.1: Set of chromatic, textural, and geometric descriptors used to represent fruit images.

Feature	Description
R, G, B	mean intensity of the red, green, and blue channels
NC	mean intensity in grayscale
LBP	texture descriptor (Local Binary Pattern)
HOG	shape descriptor (Histogram of Oriented Gradients)
Area	number of pixels in the object
Perimeter	contour length
Circularity	measure of how circular the object is
Width/height ratio	proportion between the width and height of the region

In this example, each image is represented by the vector

$$X = (R, G, B, \text{LBP}, \text{HOG}, \text{Área}, \text{Perímetro}, \text{Circularidade}, \text{Razão}).$$

Simplification adopted in this table

In practice, LBP and HOG are not scalar values but histograms with dozens or hundreds of components. In this section, each of them is represented by a single value only to simplify the presentation. In real applications, these positions would be replaced by the full components of the respective histograms.

In real applications, not all descriptors contribute equally to distinguishing the classes. Some provide more relevant information, while others may be redundant or poorly discriminative.

To reproduce this scenario in a controlled manner, `make_classification()` from the `scikit-learn` library will be used. The function generates a synthetic dataset whose features can be interpreted as image descriptors, allowing one to define how many of them will be informative for classification.

In this example, ten synthetic features are generated, of which only seven (`n_informative=7`) participate in the separation among the three classes. The remaining ones simulate poorly informative or redundant attributes. Figure 7.10 presents a conceptual representation of this process.

Before introducing the algorithm that will be studied in detail in this chapter, a remark is in order: to identify, among the ten synthetic features, which ones are most discriminative — and thus select two of them for 2D visualization — a *Random Forest* is used only as an auxiliary diagnostic tool. The KNN, the focus of this chapter, is presented next.

```
# Configuration for reproducibility
np.random.seed(42)

# Generation of synthetic data with controlled features
X, y = make_classification(
    n_samples=300,
    n_features=10,
    n_informative=7,
    n_redundant=2,
    n_repeated=1,        # One feature is a copy of another
```

```
    n_classes=3,
    n_clusters_per_class=1,
    random_state=42,
)

# Create figure with two subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# Subplot 1: Visualization of classes in 2D (using two informative features)
# Identify which features are most informative
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X, y)
importancias = rf.feature_importances_
caracteristicas_informativas = np.argsort(importancias)[-2:]  # Two most important

cores = {0: "#e74c3c", 1: "#f1c40f", 2: "#e67e22"}  # Apple, Banana, Orange
rotulos = {0: "Maçã", 1: "Banana", 2: "Laranja"}

for classe in range(3):
    idx = y == classe
    ax1.scatter(X[idx, caracteristicas_informativas[0]],
                X[idx, caracteristicas_informativas[1]],
                c=cores[classe], label=rotulos[classe],
                alpha=0.6, s=50, edgecolors="white", linewidth=0.5)

ax1.set_xlabel(f"Característica {caracteristicas_informativas[0]+1} (informativa)",
fontsize=11)
ax1.set_ylabel(f"Característica {caracteristicas_informativas[1]+1} (informativa)",
fontsize=11)
ax1.set_title("Classes no Espaço de Características\n(2 características informativas)",
              fontsize=12, fontweight="bold")
ax1.legend(loc="upper right")
ax1.grid(alpha=0.3)

# Subplot 2: Feature importance
bars = ax2.bar(range(1, 11), importancias, color="#4a90d9", alpha=0.7)
ax2.set_xlabel("Índice da Característica", fontsize=11)
ax2.set_ylabel("Importância", fontsize=11)
ax2.set_title("Importância de cada Característica\npara a Classificação",
              fontsize=12, fontweight="bold")
ax2.set_xticks(range(1, 11))
ax2.grid(axis="y", alpha=0.3)

# Color bars to highlight features
cores_barras = ["#e74c3c" if i < 7 else "#95a5a6" for i in range(10)]
for bar, cor in zip(bars, cores_barras):
    bar.set_color(cor)

# Add legend
from matplotlib.patches import Patch
legenda_elements = [
    Patch(facecolor="#e74c3c", label="Características Informativas (7)"),
    Patch(facecolor="#95a5a6", label="Características Redundantes (3)")
]
ax2.legend(handles=legenda_elements, loc="upper right")

# Annotate the number of informative features
ax2.axhline(y=0.15, color="red", linestyle="--", alpha=0.3)
ax2.text(0.5, 0.17, "Limiar de importância", fontsize=9, color="red", alpha=0.7)

plt.tight_layout()
```

```
plt.show()

print("\n Analysis of the generated data:")
print(f"  • Total samples: {X.shape[0]}")
print(f"  • Number of features: {X.shape[1]}")
print(f"  • Informative features: 7 (columns 1 to 7 of the chart)")
print(f"  • Redundant features: 2 (columns 8 and 9)")
print(f"  • Duplicated features: 1 (column 10)")
print(f"  • Class distribution: {np.bincount(y)}")
```

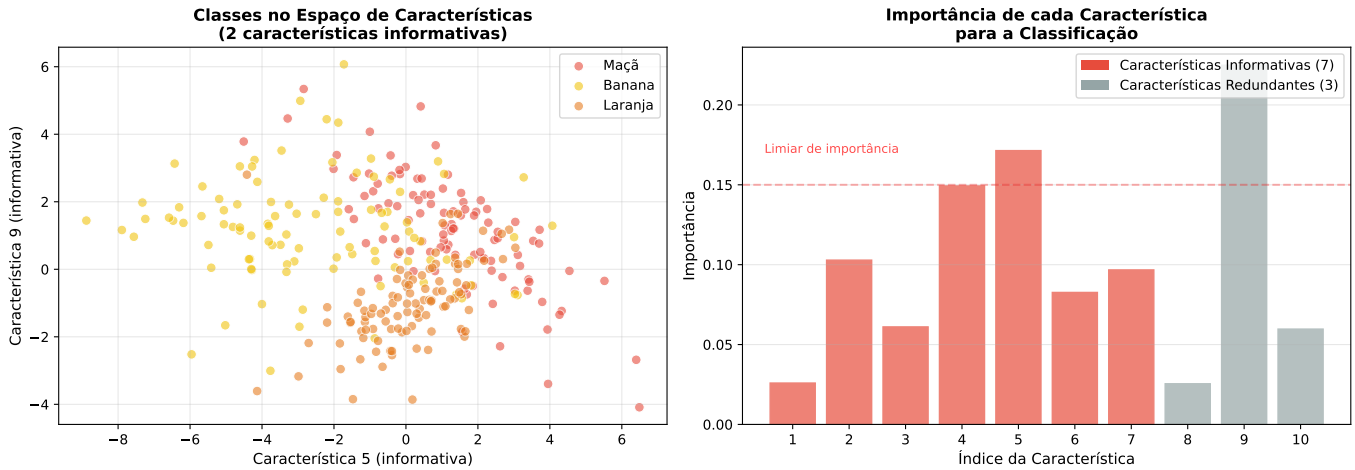


Figure 7.10: Illustration of the synthetic data generation process with *make_classification*. On the left, visualization of the three classes in a two-dimensional space formed by two informative features. On the right, relative importance of each feature for classification, showing that only 7 of the 10 features are effectively discriminative, while the others are redundant (2) or duplicated (1).

```
Analysis of the generated data:
 • Total samples: 300
 • Number of features: 10
 • Informative features: 7 (columns 1 to 7 of the chart)
 • Redundant features: 2 (columns 8 and 9)
 • Duplicated features: 1 (column 10)
 • Class distribution: [101  98 101]
```

7.10 k-NN Classifier: How It Works Internally

The **k-Nearest Neighbors (k-NN)** is one of the simplest and most intuitive classification algorithms in Machine Learning. Unlike many classifiers, it does not explicitly build a model during the training stage. Instead, it stores the labeled samples and, when a new sample needs to be classified, it looks for those that most resemble it.

The algorithm's principle is based on the hypothesis that samples with similar characteristics tend to belong to the same class. To quantify this proximity, k-NN uses a distance measure between feature vectors.

As an example, consider Table 7.2, which presents a simplified version of the fruit classification problem using only two features: color intensity and circularity, both normalized in the range from 0 to 1.

Table 7.2: Simplified example of fruit classification using two normalized features.

Sample	Color	Circularity	Class
Fruit 1	0.82	0.88	Apple
Fruit 2	0.30	0.20	Banana
Fruit 3	0.88	0.85	Apple
Fruit ? (test)	0.80	0.90	?

Observing only these two features, one notices that the test fruit is much closer to the samples labeled as **Apple** than to the sample labeled as **Banana**. In the following section, this intuitive notion of proximity will be formalized through a distance metric, used by the algorithm to identify the nearest neighbors and decide the class of the new sample.

7.10.1 Distance Metric

The proximity between two samples is typically quantified by the **Euclidean distance**, defined as

$$d(x, x_i) = \|x - x_i\|_2 = \sqrt{\sum_{j=1}^{n} (x_j - x_{i,j})^2},$$

where:

- x is the test sample;
- x_i is a sample from the training set;
- n is the number of features;
- x_j and $x_{i,j}$ represent the j-th feature.

In the implementation of this chapter, x corresponds to a row of `X_test` and x_i to a row of `X_train`. The `predict()` method automatically computes the distance between x and all training samples.

In the example from Table 7.2:

$$d(\text{test}, \text{Fruit 1}) \approx 0{,}028, \qquad d(\text{test}, \text{Fruit 2}) \approx 0{,}860, \qquad d(\text{test}, \text{Fruit 3}) \approx 0{,}094.$$

Since the smallest distances correspond to Fruits 1 and 3, these samples will be used in the decision step.

7.10.2 Decision Rule

After sorting the distances, the algorithm selects the k nearest neighbors. Let $N_k(x)$ be this set. The predicted class is given by

$$\hat{y} = \text{mode}\{\, y_i : x_i \in N_k(x) \,\},$$

where y_i is the label of sample x_i and $\hat{y}$ is the class assigned to the test sample.

In the example, for $k = 3$, the neighbors are Fruit 1 (Apple), Fruit 3 (Apple), and Fruit 2 (Banana). Since **Apple** receives two votes, this is the predicted class.

`KNeighborsClassifier` Class

In this chapter, the algorithm is implemented using the `KNeighborsClassifier` class from the `scikit-learn` library:

```python
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

y_pred = knn.predict(X_test)
```

where:

- `KNeighborsClassifier(n_neighbors=3)`: sets the value of k;
- `fit(X_train, y_train)`: stores the training samples (`X_train`) and their labels (`y_train`);
- `predict(X_test)`: returns the predicted classes for the samples in `X_test`.

Internally, `predict()` performs the steps described earlier: it calculates the distances, identifies the k nearest neighbors, and determines the class by majority voting.

Figure 7.11 illustrates this procedure on a two-dimensional dataset. The figure highlights the neighbors used in the classification, while the console displays the algorithm's steps: distance calculation, sorting, neighbor selection, voting, and class prediction.

```python
def knn_passo_a_passo(X, y, x, k=3):
    """Runs the five steps of the k-NN algorithm.
    Parameters: X (training), y (labels), x (test), and k (number of neighbors).
    """
    # 1. Distances
    dist = [(np.linalg.norm(x-xi), yi, i) for i, (xi, yi) in enumerate(zip(X, y))]
    print(f"1. Distances computed: {len(dist)}")

    # 2. Sorting
    dist.sort(key=lambda t: t[0])
    print("2. Sorted distances")

    # 3. Selection
    vizinhos = dist[:k]
    print(f"3. {k} nearest neighbors:")
    for d, c, _ in vizinhos:
        print(f"   {d:.4f} → {c}")

    # 4. Voting
    votos = {}
    for _, c, _ in vizinhos:
        votos[c] = votos.get(c, 0) + 1
    print("4. Votes:", votos)

    # 5. Decision
    classe = max(votos, key=votos.get)
    print("5. Predicted class:", classe)

    return classe, vizinhos


# Example data
np.random.seed(4)
X = np.r_[np.random.randn(15,2)+[2,2],
          np.random.randn(15,2)+[-2,-2]]
y = np.array(["Classe A"]*15 + ["Classe B"]*15)
x = np.array([0.5,0.5])

classe, vizinhos = knn_passo_a_passo(X, y, x)

# Visualization
```

```python
plt.figure(figsize=(5,5))

for c, rotulo in [("Classe A","Classe A"), ("Classe B","Classe B")]:
    P = X[y==c]
    plt.scatter(P[:,0], P[:,1], s=80, label=rotulo)

plt.scatter(*x, marker="*", s=220, edgecolors="black", label="Teste")

for _, _, i in vizinhos:
    plt.scatter(*X[i], s=220, facecolors="none", edgecolors="black", linewidths=2)
    plt.plot([x[0], X[i,0]], [x[1], X[i,1]], "--", lw=1)

plt.xlabel("Característica 1")
plt.ylabel("Característica 2")
plt.title(f"k-NN ($k=3$): classe predita = {classe}")
plt.legend()
plt.grid(alpha=.3)
plt.axis("equal")
plt.tight_layout()
plt.show()
```

```
1. Distances computed: 30
2. Sorted distances
3. 3 nearest neighbors:
   1.0850 → Classe A
   1.6379 → Classe B
   1.6382 → Classe A
4. Votes: {np.str_('Classe A'): 2, np.str_('Classe B'): 1}
5. Predicted class: Classe A
```

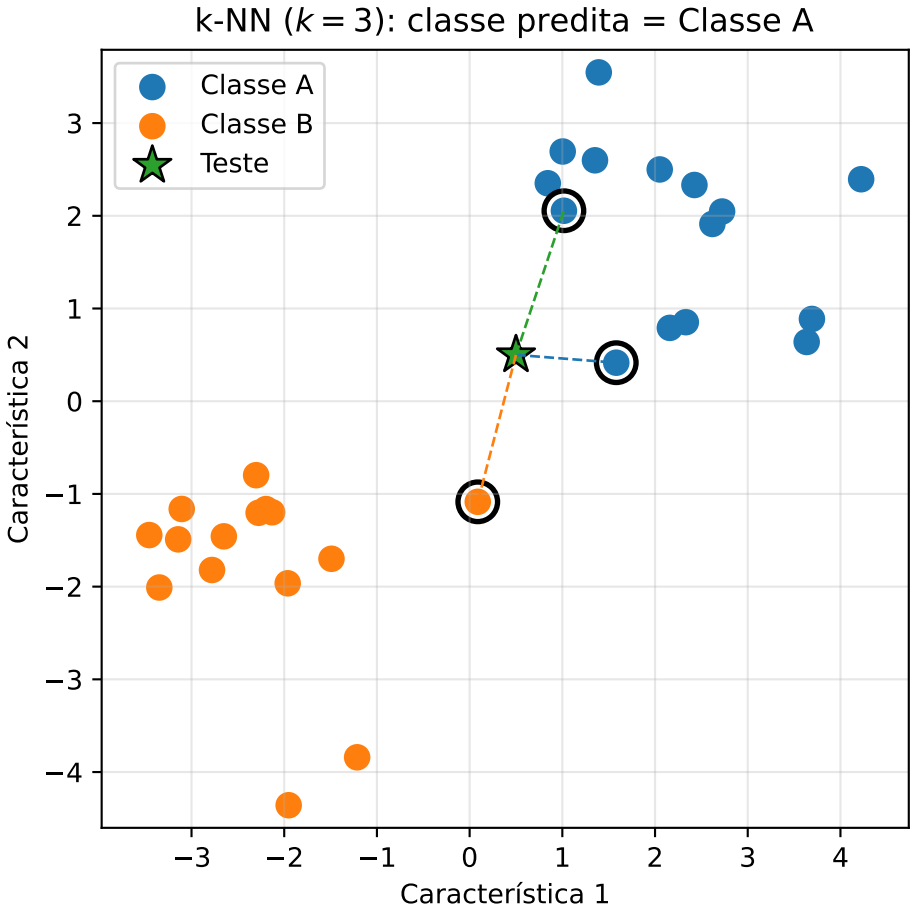


Figure 7.11: Classification of a new sample by the k-NN algorithm. The test point (star) is classified using the three nearest neighbors, highlighted by circles.

7.10.3 The Role of the Parameter k

The parameter k determines how many neighbors participate in the classification decision.

- **Small values of** k (for example, $k = 1$) make the classifier more sensitive to noise and local variations, producing more irregular decision boundaries and favoring *overfitting*.
- **Larger values of** k produce smoother decision boundaries, but may reduce sensitivity to local structures, favoring *underfitting*.

In problems with two classes, it is common to use odd values of k to reduce the occurrence of ties.

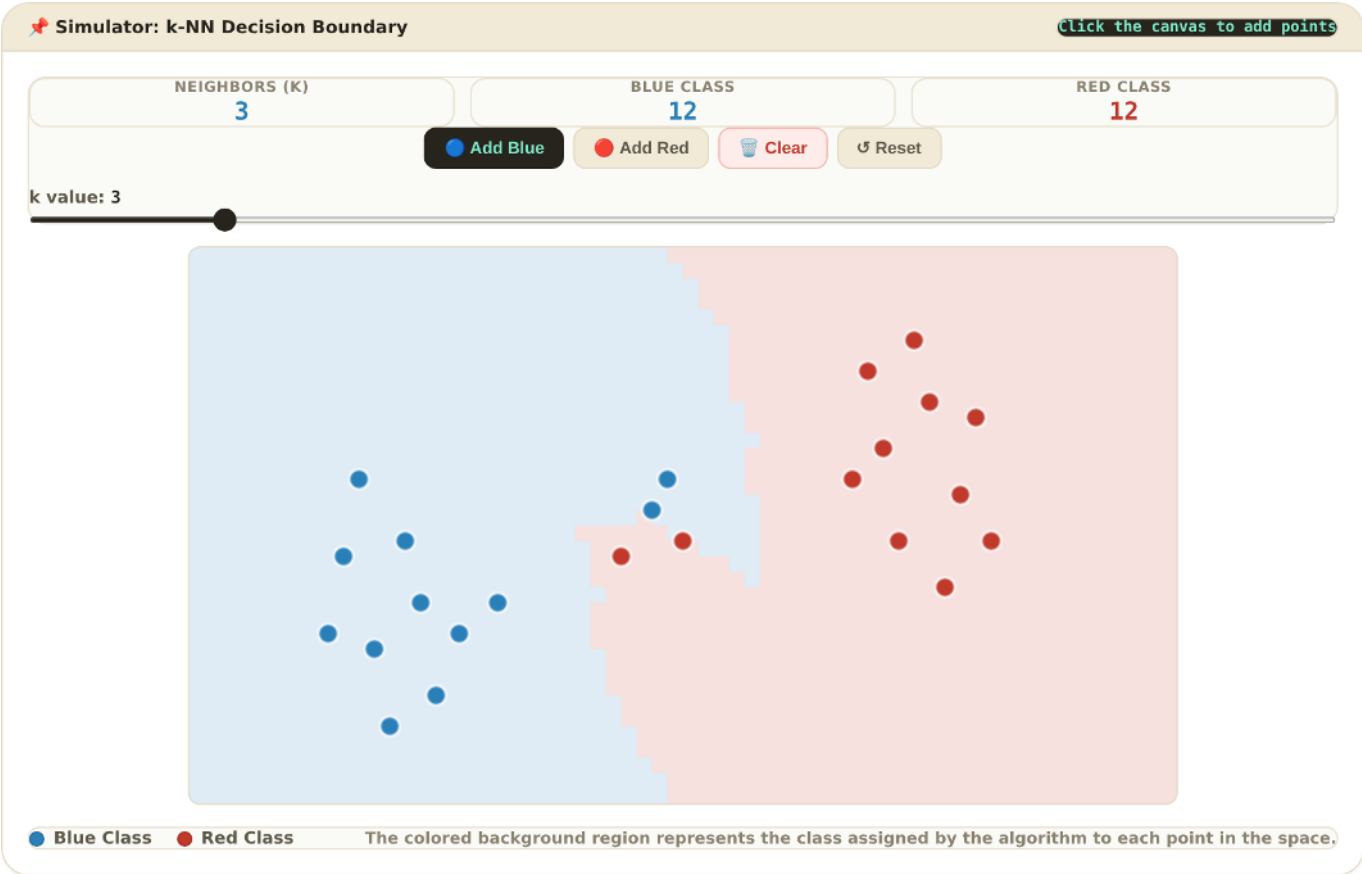
Another important aspect is the **curse of dimensionality**. As the number of features increases, the distances between samples tend to become more similar, making it difficult to identify truly representative neighbors.

Summary

The k-NN algorithm can be summarized in three steps:

1. extract the feature vector of the new sample;
2. identify the k nearest neighbors;
3. classify the sample by the most frequent class among those neighbors.

The simulator in Figure 7.12 allows you to visually explore the effect of the parameter k on the decision boundary.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap07/sim-07-knn.html

Figure 7.12: Interactive simulator of the k-NN decision boundary: add training points and adjust the value of k to observe the effect on the decision region.

7.11 Practical Project 1: Handwritten Digit Classification with k-NN

The previous sections presented the k-NN algorithm through a simplified fruit classification example, using only two features. Next, the same algorithm is applied to an image dataset, in which each sample is represented by a higher-dimensional vector.

As a case study, the public `load_digits` database, provided by the `scikit-learn` library, is used. This dataset contains 1797 images of handwritten digits from classes 0 to 9, each with a resolution of 8×8 pixels in grayscale. Each image is represented by a vector with 64 features, corresponding to pixel intensities, and each vector has a label indicating the corresponding digit.

The `load_digits` database is provided by the `scikit-learn` library and is used in this chapter to illustrate the application of the k-NN algorithm. In addition to being available directly in `scikit-learn`, it eliminates the need for additional steps of data acquisition and preparation, allowing attention to be focused on the implementation and evaluation of the classifier.

Figure 7.13 presents a sample of the images from the database.

```
digits = load_digits()
print(f"Total samples: {digits.data.shape[0]}, vector dimension: {digits.data.shape[1]}")
print(f"Classes: {[int(i) for i in sorted(set(digits.target))]}")

n_amostras = 16
imgs = list(digits.images[:n_amostras])
imgs_titles = [str(label) for label in digits.target[:n_amostras]]
mm.show(imgs, titles=imgs_titles, cols=8, figsize=(12, 4))
```

```
Total samples: 1797, vector dimension: 64
Classes: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

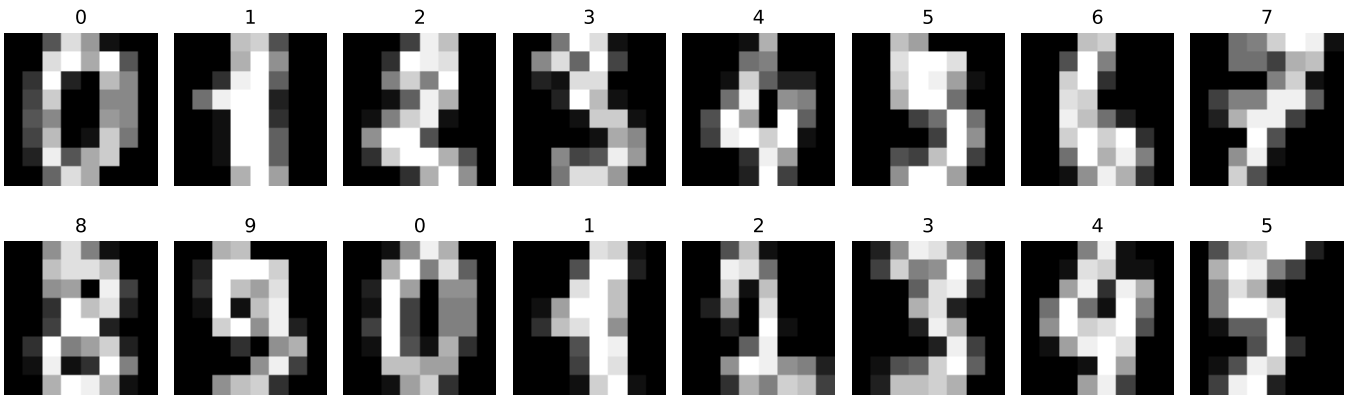


Figure 7.13: Sample of handwritten digits from the *load_digits* dataset, used as a classification case study.

7.11.1 Classification with Intensity Vectors

In this first experiment, each 8×8 image is represented directly by the intensities of its 64 pixels, without extracting additional descriptors. Thus, each sample corresponds to a vector of 64 features, used as input to the k-NN classifier.

Next, the dataset is divided into training and test subsets, preserving the proportion of the ten classes through the parameter `stratify=y`. The classifier is trained with $k = 3$ and evaluated on the test set using accuracy and the confusion matrix shown in Figure 7.14.

```
X, y = digits.data, digits.target

X_treino, X_teste, y_treino, y_teste = train_test_split(
```

```
    X, y, test_size=0.3, random_state=42, stratify=y
)

n_neighbors = 3
knn_pixels = KNeighborsClassifier(n_neighbors=n_neighbors)
knn_pixels.fit(X_treino, y_treino)
pred_pixels = knn_pixels.predict(X_teste)

acc_pixels = accuracy_score(y_teste, pred_pixels)
print(f"Accuracy (intensity vectors, k={n_neighbors}): {acc_pixels:.4f}")

cm = confusion_matrix(y_teste, pred_pixels)

plt.figure(figsize=(5,4))
plt.imshow(cm, cmap="Blues")

for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(
            j, i, cm[i, j],
            ha="center", va="center",
            color="white" if cm[i, j] > cm.max()/2 else "black",
            fontsize=9
        )

plt.title("Matriz de Confusão - Pixels Brutos")
plt.xlabel("Classe Predita")
plt.ylabel("Classe Real")
plt.xticks(range(10))
plt.yticks(range(10))
plt.colorbar(fraction=0.046)
plt.tight_layout()
plt.show()
```

```
Accuracy (intensity vectors, k=3): 0.9870
```

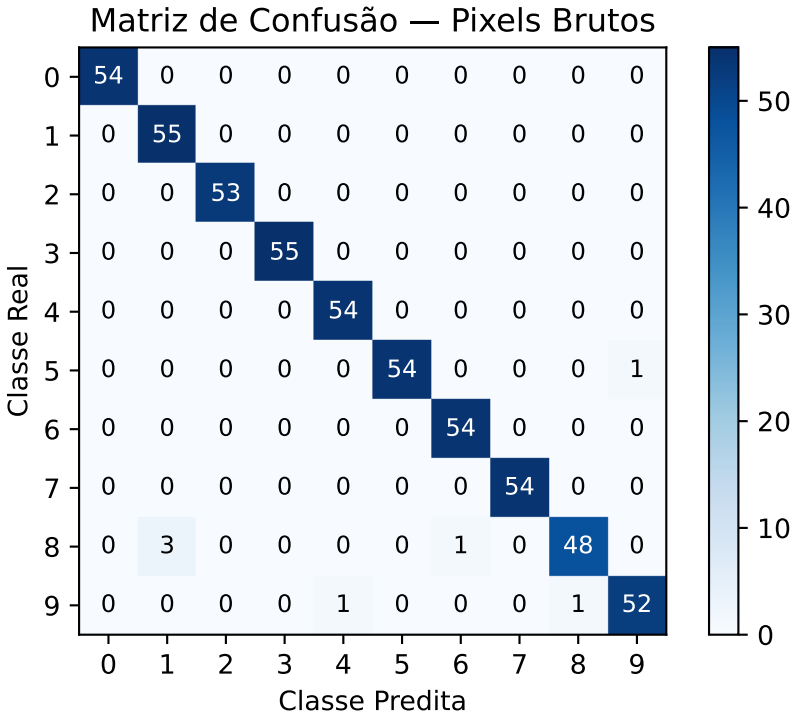


Figure 7.14: Confusion matrix of the k-NN classifier trained with raw intensity vectors (pixels).

7.11.2 Classification with HOG Descriptors

In the previous experiment, each image was represented directly by the intensities of its pixels. In this section, this representation is replaced by HOG descriptors (*Histogram of Oriented Gradients*), which encode information about the distribution of gradient orientations in the image.

The same data partitioning, the same k-NN classifier, and the same evaluation protocol are maintained, with only the image representation being changed. Figure 7.15 compares the results obtained with intensity vectors and with HOG descriptors.

```
descritores_hog = np.array([
    hog(img, orientations=8, pixels_per_cell=(4, 4), cells_per_block=(1, 1))
    for img in digits.images
])
print(f"HOG vector dimension: {descritores_hog.shape[1]}")

Xh_treino, Xh_teste, yh_treino, yh_teste = train_test_split(
    descritores_hog, y, test_size=0.3, random_state=42, stratify=y
)

n_neighbors = 3
knn_hog = KNeighborsClassifier(n_neighbors=n_neighbors)
knn_hog.fit(Xh_treino, yh_treino)
pred_hog = knn_hog.predict(Xh_teste)
acc_hog = accuracy_score(yh_teste, pred_hog)
print(f"Accuracy (HOG descriptor, k={n_neighbors}): {acc_hog:.4f}")

plt.figure(figsize=(4, 3))
plt.bar(["Pixels brutos", "HOG"], [acc_pixels, acc_hog], color=["#6366f1", "#f97316"])
plt.ylim(0, 1.1)  # Increases the upper limit to give space
plt.ylabel("Acurácia")
plt.title("Comparação de Descritores")
for i, v in enumerate([acc_pixels, acc_hog]):
```

```
    plt.text(i, v + 0.02, f"{v:.3f}", ha="center")  # Increases the vertical offset
plt.tight_layout()
```

```
HOG vector dimension: 32
Accuracy (HOG descriptor, k=3): 0.7593
```

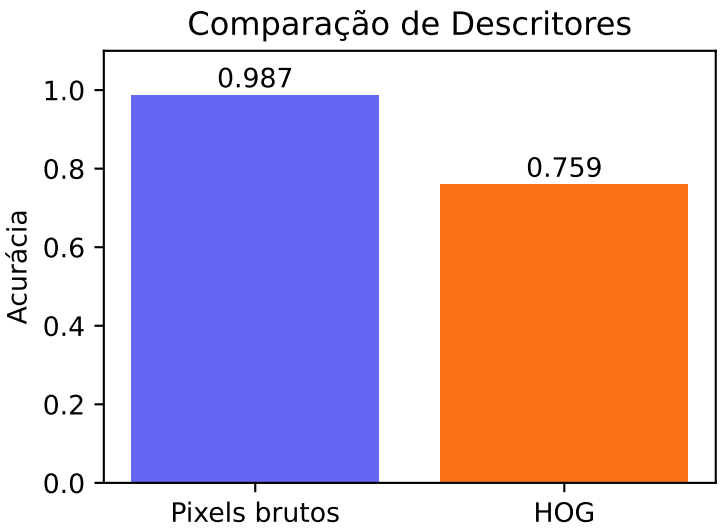


Figure 7.15: Comparison of accuracy between raw pixel descriptors and HOG for the k-NN classifier (k=3) on the digits dataset.

Why does this happen?

In Figure 7.15, the classifier trained with **pixel intensity** vectors achieves higher accuracy (0.987) than the one based on **HOG** descriptors (0.759). This result is related to the characteristics of the `load_digits` dataset.
The images have a resolution of only 8×8 pixels, are approximately centered, and exhibit little variation in illumination, scale, and orientation. In this scenario, pixel intensities preserve virtually all the information needed to distinguish the classes. In contrast, HOG summarizes the image into histograms of gradient orientations, reducing part of the spatial detail available in the original pixels.
This reduction in information can make it harder to separate visually similar digits, such as 3 and 8 or 4 and 9, especially when the image resolution is low.
In problems with higher-resolution images or those subject to variations in illumination, position, scale, or small deformations, descriptors such as HOG tend to better represent the local structure of the image than individual pixel values. Thus, this experiment illustrates an important principle of Machine Learning: **the data representation should be chosen according to the characteristics of the problem, not by the complexity of the descriptor.**

7.12 Classifier Evaluation

In the previous sections, the quality of the classifier was analyzed through accuracy and the confusion matrix. In this section, these tools are complemented by metrics used in model evaluation and by a procedure for selecting the value of the parameter k.

Accuracy corresponds to the proportion of correctly classified samples. Although it is a simple and widely used measure, it may be insufficient when classes exhibit highly imbalanced distributions.

From the confusion matrix — introduced in **Chapter 1** and used throughout this chapter — class-specific metrics can be calculated, such as precision and recall:

$$\text{Precision} = \frac{TP}{TP + FP}, \qquad \text{Recall} = \frac{TP}{TP + FN},$$

where TP, FP, and FN represent, respectively, the number of true positives, false positives, and false negatives for the class under analysis. Precision quantifies the proportion of correct positive predictions, while recall measures the classifier's ability to identify examples belonging to the class.

7.12.1 Choosing k by Cross-Validation

In the previous experiments, $k = 3$ was adopted to illustrate how the algorithm works. However, this parameter directly influences the classifier's performance and, in practice, should be selected based on the data.

A widely used approach is **cross-validation**, in which the training set is divided into successive partitions to estimate the model's performance on data not used during training.

The following code computes the average accuracy obtained by five-fold cross-validation for different values of k. The results are presented in Figure 7.16, allowing the identification of the region where the classifier achieves its best performance.

```
valores_k = range(1, 16)
acuracias_medias = []

for k in valores_k:
    modelo = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(modelo, X, y, cv=5)
    acuracias_medias.append(scores.mean())

melhor_k = list(valores_k)[int(np.argmax(acuracias_medias))]
print(f"Best k value found: {melhor_k} (mean accuracy={max(acuracias_medias):.4f})")

plt.figure(figsize=(6, 4))
plt.plot(list(valores_k), acuracias_medias, marker="o", color="#4f46e5")
plt.axvline(melhor_k, color="#f97316", linestyle="--", label=f"melhor k = {melhor_k}")
plt.xlabel("k")
plt.ylabel("Acurácia média (validação cruzada)")
plt.title("Seleção de k por Validação Cruzada")
plt.legend()
plt.grid(alpha=0.3)
plt.tight_layout()
```

```
Best k value found: 2 (mean accuracy=0.9672)
```

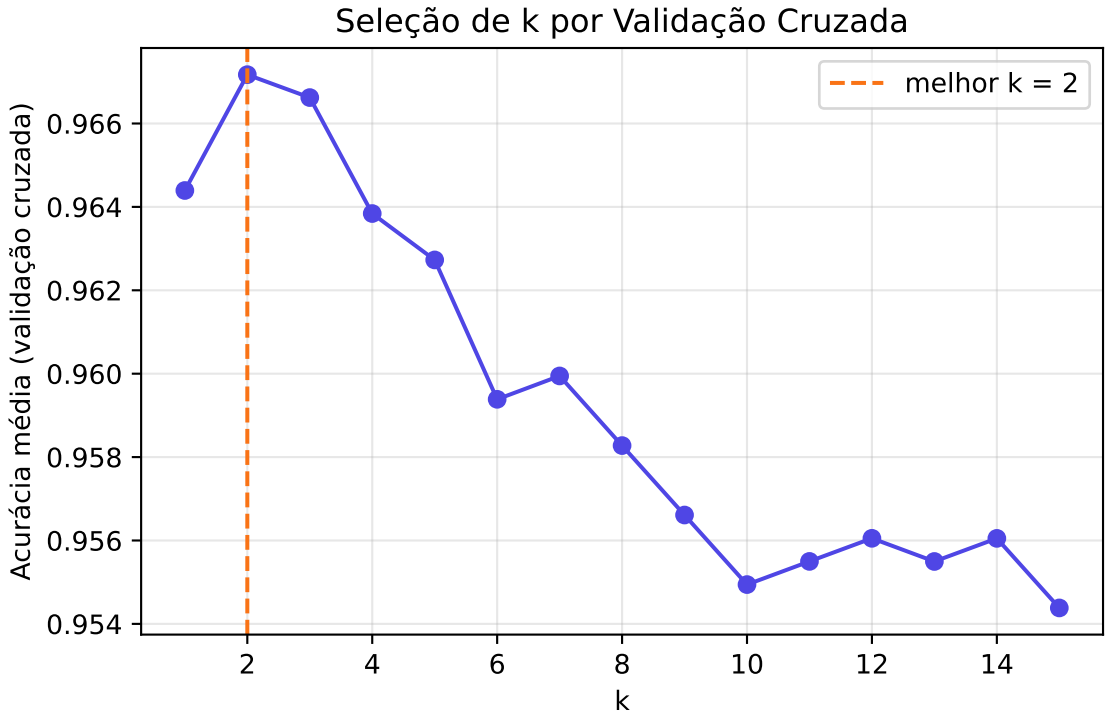


Figure 7.16: Mean accuracy by cross-validation (5 folds) as a function of parameter k, for the digit base with raw intensity vectors.

7.12.2 The Bias-Variance Trade-off: Diagnosing *Overfitting* and *Underfitting*

The hyperparameter k influences the complexity of the k-NN classifier's decision boundary and, consequently, its generalization capability. In general terms, small values of k make the model more sensitive to training samples, while larger values produce smoother decision boundaries.

These behaviors are associated with the trade-off between **bias** and **variance**. Very small values of k tend to increase the risk of **overfitting**, especially in noisy datasets, whereas very large values can lead to **underfitting**, reducing the model's ability to capture local data structures.

While Figure 7.16 presented only the average accuracy obtained through cross-validation, Figure 7.17 compares training and test accuracies for different values of k. The highlighted regions in the plot represent the algorithm's expected behavior: a higher risk of overfitting for small values of k, an intermediate region that often yields a good balance between bias and variance, and a higher risk of underfitting for large values of k.

However, these regions should be interpreted only as a conceptual reference. The observed behavior depends on the characteristics of the dataset. In the `load_digits` dataset, for example, the images exhibit low variability and good class separation, so small values of k may perform similarly—or even better—than others, without showing significant overfitting.

```
k_values = range(1, 16)
train_acc, test_acc = [], []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k).fit(X_treino, y_treino)
    train_acc.append(accuracy_score(y_treino, knn.predict(X_treino)))
    test_acc.append(accuracy_score(y_teste, knn.predict(X_teste)))

plt.figure(figsize=(9,5))
plt.plot(k_values, train_acc, "o-", lw=2, label="Treinamento")
plt.plot(k_values, test_acc,  "s-", lw=2, label="Teste")
```

```
plt.axvspan(1, 3,  color="#fca5a5", alpha=.25, label="Maior risco de overfitting")
plt.axvspan(3,11,  color="#86efac", alpha=.25, label="Compromisso entre viés e variância")
plt.axvspan(11,15, color="#93c5fd", alpha=.25, label="Maior risco de underfitting")

plt.xlabel("Número de vizinhos ($k$)")
plt.ylabel("Acurácia")
plt.xticks(k_values)
plt.grid(alpha=.3)
plt.legend(loc="upper right")
plt.tight_layout()
plt.show()

maior_acc = max(test_acc)
melhores_k = [k for k, a in zip(k_values, test_acc) if np.isclose(a, maior_acc)]

print("Interpretation")
print("- Small k values: higher risk of overfitting.")
print("- Intermediate values: better bias-variance trade-off.")
print("- Large k values: higher risk of underfitting.")
print("\nThe colored regions represent general trends;")
print("the observed behavior depends on the dataset.")
print(f"\nHighest test accuracy: {maior_acc:.3f}")
print(f"k values that achieved this accuracy: {melhores_k}")
```

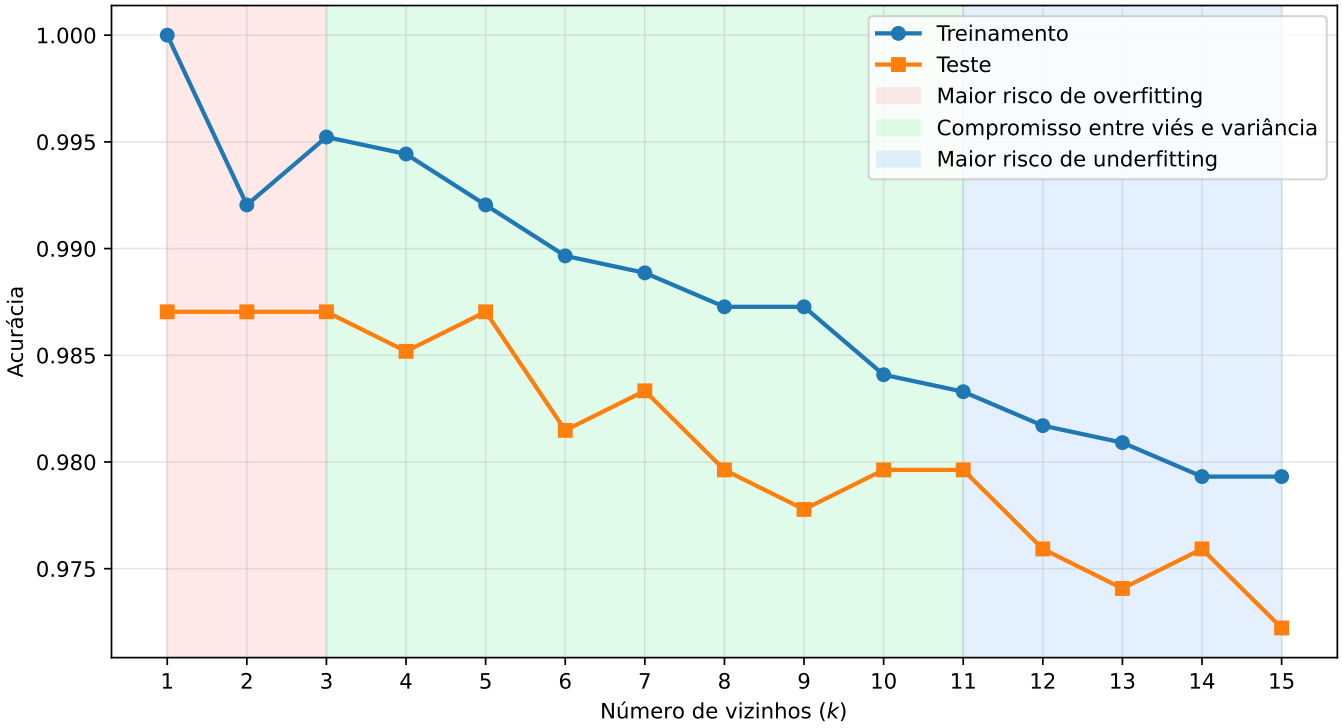


Figure 7.17: Accuracy on the training and test sets for different values of k. The colored regions represent, conceptually, behavior trends of the classifier: higher risk of overfitting (red), bias-variance trade-off (green) and higher risk of underfitting (blue).

```
Interpretation
- Small k values: higher risk of overfitting.
- Intermediate values: better bias-variance trade-off.
- Large k values: higher risk of underfitting.

The colored regions represent general trends;
```

```
the observed behavior depends on the dataset.

Highest test accuracy: 0.987
k values that achieved this accuracy: [1, 2, 3, 5]
```

7.13 Practical Project 2: Comparison of Descriptors for Texture Classification

In **Chapter 6**, local variance was used as a texture descriptor to **detect** anomalies in industrial surfaces, distinguishing **conforming** and **defective** samples. In this project, the problem is reformulated as a **multiclass classification** task, in which different image representations are used as input to a classifier.

Three types of descriptors will be considered: **pixel intensities**, **Local Binary Patterns (LBP)**, and **Histogram of Oriented Gradients (HOG)**. For each representation, a feature vector will be extracted to serve as input for the k-nearest neighbors (k-NN) algorithm. Finally, the accuracies obtained by each descriptor under the conditions defined for this experiment will be compared.

The evaluation will be carried out through **five-fold stratified cross-validation**. In this procedure, the dataset is divided into five subsets while preserving the proportion between classes. In each iteration, one partition is used for testing and the remaining four for training, repeating the process until all partitions have been used as the test set. At the end of the five runs, the mean accuracy and standard deviation are calculated for each descriptor.

The dataset consists of three classes of synthetic textures: **granular**, obtained from smoothed Gaussian noise; **striped**, formed by periodic sinusoidal patterns; and **blotched**, composed of overlapping circular regions. To introduce variability among samples, all images receive a low-intensity Gaussian noise perturbation. Figure 7.18 presents examples of the three classes used in the experiment.

```python
rng = np.random.default_rng(42)

def gerar_textura(classe, tamanho=64, ruido=0.10):
    """Generates a synthetic 64x64 texture belonging to one of the three classes."""
    if classe == "granular":
        escala = rng.uniform(0.14, 0.22)
        img = rng.normal(0.5, escala, (tamanho, tamanho))
        img = cv2.GaussianBlur(img.astype(np.float32), (3, 3), 0)

    elif classe == "listrada":
        n_periodos = rng.uniform(4, 8)
        amplitude = rng.uniform(0.22, 0.38)
        eixo_x = np.linspace(0, n_periodos * np.pi, tamanho)
        base = 0.5 + amplitude * np.sin(eixo_x)
        img = np.tile(base, (tamanho, 1)).astype(np.float32)
        img += rng.normal(0, 0.09, (tamanho, tamanho)).astype(np.float32)

    elif classe == "manchada":
        img = np.full((tamanho, tamanho), 0.5, dtype=np.float32)
        n_manchas = rng.integers(5, 11)
        for _ in range(n_manchas):
            cx, cy = rng.integers(0, tamanho, 2)
            raio = int(rng.integers(3, 11))
            intensidade = float(rng.uniform(0.15, 0.9))
            cv2.circle(img, (int(cx), int(cy)), raio, intensidade, -1)
        img = cv2.GaussianBlur(img, (5, 5), 0)

    else:
        raise ValueError(f"Classe desconhecida: {classe}")
```

```
    img = img + rng.normal(0, ruido, (tamanho, tamanho)).astype(np.float32)
    img = np.clip(img, 0, 1)
    return (img * 255).astype(np.uint8)

classes_textura = ["granular", "listrada", "manchada"]
amostras = [gerar_textura(c) for c in classes_textura]

mm.show(amostras, titles=classes_textura, cols=3, figsize=(9, 3))
```

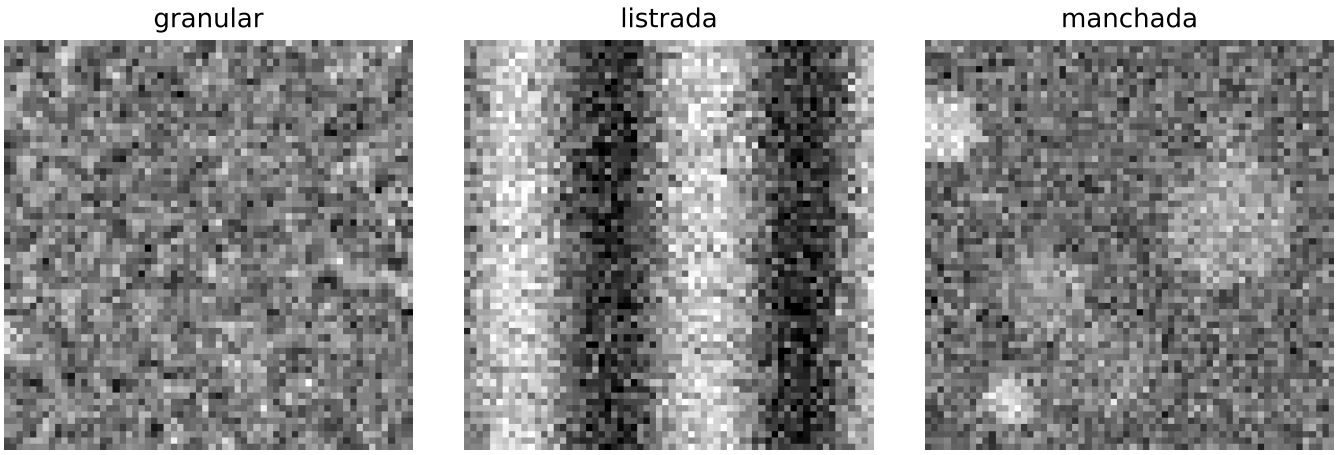


Figure 7.18: Synthetic samples of the three texture classes used in the classification experiment, generated with Gaussian noise and intra-class variability.

7.13.1 Feature Extraction Pipeline and Comparative Evaluation

To compare the performance of different image representation approaches, a balanced dataset containing 60 samples per class was generated. From this dataset, three types of feature vectors were extracted, each representing distinct aspects of visual information:

1. **Raw pixels**: vector obtained by flattening the image intensity matrix, resulting in a vector of $64 \times 64 = 4096$ attributes;
2. **Uniform LBP histogram**: normalized histogram of the frequencies of local patterns produced by the uniform LBP operator, comprising 10 attributes;
3. **HOG descriptor**: vector formed by histograms of oriented gradients, which represent the spatial distribution of edge orientations, totaling 128 attributes.

Since these descriptors have distinct scales and dimensionalities, the feature vectors are standardized using the `StandardScaler`, so that each attribute has zero mean and unit standard deviation. This step prevents attributes with larger magnitudes from disproportionately influencing the Euclidean distance calculation employed by the classifier.

The evaluation is performed using the k-NN algorithm with $k = 5$, under the same stratified five-fold cross-validation protocol described in the previous section. The average accuracy obtained across the five runs, see Figure 7.19, provides a more stable estimate of the classifier's performance, reducing the dependence on a single training–test split.

```
rng = np.random.default_rng(42)

# 1. Dataset generation
X_bruto, X_lbp, X_hog, y_textura = [], [], [], []

for classe in classes_textura:
    for _ in range(60):
```

```
        img = gerar_textura(classe, ruido=0.10)

        # Extraction 1: Raw pixels
        X_bruto.append(img.ravel())

        # Extraction 2: Uniform LBP histogram
        lbp = local_binary_pattern(img, P=8, R=1, method="uniform")
        hist_lbp, _ = np.histogram(lbp, bins=10, range=(0, 10), density=True)
        X_lbp.append(hist_lbp)

        # Extraction 3: HOG descriptor
        feat_hog = hog(img, orientations=8, pixels_per_cell=(16, 16), cells_per_block=(1, 1))
        X_hog.append(feat_hog)

        y_textura.append(classe)

y_textura = np.array(y_textura)
descritores = {
    "Pixels Brutos": np.array(X_bruto),
    "LBP (Textura)": np.array(X_lbp),
    "HOG (Forma)": np.array(X_hog)
}

# 2. Statistical evaluation via 5-fold cross-validation
resultados_media = {}
resultados_desvio = {}

knn = KNeighborsClassifier(n_neighbors=5)
scaler = StandardScaler()

for nome, X_dados in descritores.items():
    X_norm = scaler.fit_transform(X_dados)
    scores = cross_val_score(knn, X_norm, y_textura, cv=5, scoring="accuracy")
    resultados_media[nome] = scores.mean()
    resultados_desvio[nome] = scores.std()
    print(f"{nome:15s} -> Average Accuracy: {scores.mean():.4f} (± {scores.std():.4f})")

# 3. Plotting of the formal comparative chart
plt.figure(figsize=(7, 4.5))
nomes_desc = list(resultados_media.keys())
medias = list(resultados_media.values())
desvios = list(resultados_desvio.values())

bars = plt.bar(nomes_desc, medias, yerr=desvios, capsize=6,
               color=["#6366f1", "#9333ea", "#f97316"],
               width=0.45, edgecolor="black", alpha=0.85)
plt.ylabel("Acurácia Média (5-Fold CV)", fontsize=11)
plt.title("Análise Comparativa de Descritores para Classificação de Texturas",
          fontsize=12, fontweight="bold")
plt.ylim(0.3, 1.1)
plt.grid(axis="y", linestyle="--", alpha=0.5)

for bar in bars:
    h = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2,
             h + 0.03, f"{h:.3f}", ha="center", fontweight="bold")

plt.tight_layout()
plt.show()
```

```
Pixels Brutos   -> Average Accuracy: 0.6889 (± 0.0478)
LBP (Textura)   -> Average Accuracy: 0.7611 (± 0.0648)
HOG (Forma)     -> Average Accuracy: 0.4889 (± 0.0624)
```

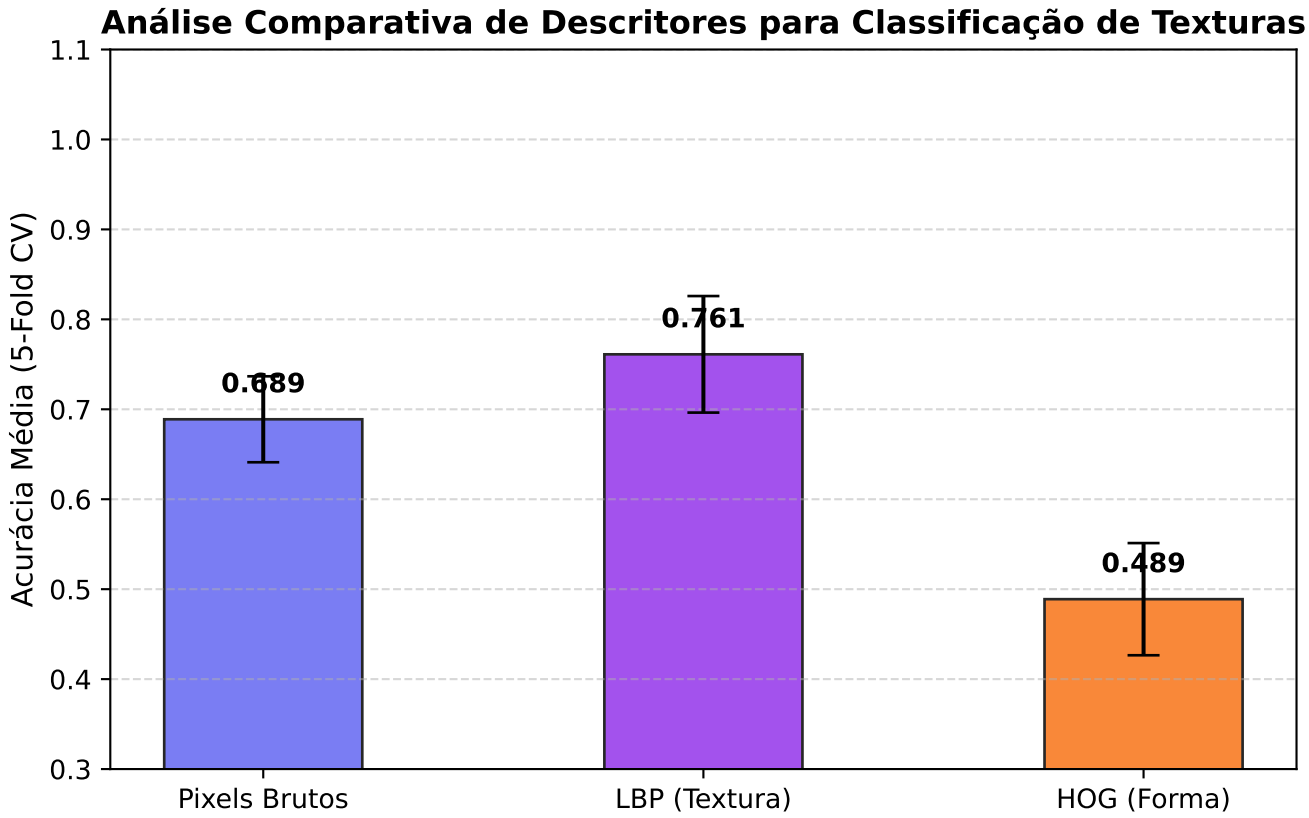


Figure 7.19: Average accuracy obtained via cross-validation (5-fold) for the descriptors of Raw Pixels, LBP, and HOG applied to the synthetic textures dataset.

Why does it work? — LBP as texture representation

The LBP descriptor represents the texture of an image through a normalized histogram that counts the frequency of local intensity patterns. Instead of directly storing pixel values or their positions, this representation summarizes the distribution of microstructures present in the image, yielding a compact feature vector.

In this experiment, three types of descriptors were compared: raw pixels, LBP, and HOG. Vectors formed by **raw pixels** preserve all image intensities, but they also incorporate variations stemming from noise and small spatial shifts, which can hinder sample comparison through Euclidean distance.

The **HOG** descriptor represents the distribution of gradient orientations, making it suitable for describing shapes and contours. Since the images used in this project differ mainly by texture properties, rather than by well-defined contours, this representation tends to capture less discriminative information than LBP.

LBP, in turn, was specifically developed to characterize local texture patterns. Its histogram describes the frequency of microstructures present in the image, regardless of their exact position, making the representation less sensitive to small spatial variations and monotonic illumination changes.

Although the histograms of different classes exhibit distinct distributions, the Gaussian noise introduced during image generation increases variability among samples within the same class and may produce overlapping regions in the feature space. As a result, some textures may be confused by the classifier. Nevertheless, when the relevant features for distinguishing classes are associated with local texture patterns, descriptors designed for this purpose, such as LBP, are expected to yield more informative

representations than those based solely on pixel intensities or gradient orientations.

7.13.2 Fine-Grained Classifier Diagnostics: Precision, Recall, and F1-Score

Accuracy summarizes the classifier's performance in a single value, but it does not indicate how this performance is distributed across different classes. For a more detailed analysis, metrics calculated individually for each class are used.

Precision measures the proportion of samples classified as belonging to a class that indeed belong to it. **Recall** measures the proportion of samples from the class that were correctly identified by the classifier. The **F1-score** corresponds to the harmonic mean of precision and recall, providing an indicator that balances both measures.

The report also provides the **support**, that is, the number of samples from each class present in the test set. This information is important for contextualizing the metrics, as results obtained from few samples tend to exhibit greater variability.

Figure 7.20 presents these metrics for the three texture classes. Together, they allow identifying performance differences that are not evident from accuracy alone. For example, a class may show high precision and lower recall, indicating that the classifier makes few false positives but fails to identify part of the samples that truly belong to that class. This type of analysis aids in understanding the model's limitations and identifying potential strategies for its improvement.

```
X_lbp_data = np.array(X_lbp)
y_textura_data = np.array(y_textura)

# Perform a train/test split for the detailed report
Xt_treino, Xt_teste, yt_treino, yt_teste = train_test_split(
    X_lbp_data, y_textura_data, test_size=0.3, random_state=42, stratify=y_textura_data
)

# Scale the data
scaler = StandardScaler()
Xt_treino_scaled = scaler.fit_transform(Xt_treino)
Xt_teste_scaled = scaler.transform(Xt_teste)

# Train the k-NN classifier
knn_textura = KNeighborsClassifier(n_neighbors=5)
knn_textura.fit(Xt_treino_scaled, yt_treino)

y_pred = knn_textura.predict(Xt_teste_scaled)

report = classification_report(
    yt_teste,
    y_pred,
    target_names=classes_textura,
    output_dict=True
)

print("=== DETAILED CLASSIFICATION REPORT ===")
print(f"{'Class':<12} {'Precision':>10} {'Recall':>12} {'F1-score':>10} {'Support':>10}")
for classe in classes_textura:
    r = report[classe]
    print(f"{classe:<12} {r['precision']:>10.2f} {r['recall']:>12.2f} "
          f"{r['f1-score']:>10.2f} {r['support']:>10.0f}")

precision = precision_score(yt_teste, y_pred, average=None, labels=classes_textura)
recall = recall_score(yt_teste, y_pred, average=None, labels=classes_textura)
f1 = f1_score(yt_teste, y_pred, average=None, labels=classes_textura)
```

```
fig, ax = plt.subplots(figsize=(10, 5))
x = np.arange(len(classes_textura))
width = 0.25

bars1 = ax.bar(x - width, precision, width, label='Precisão', color='#6366f1', alpha=0.8)
bars2 = ax.bar(x, recall, width, label='Revocação', color='#f97316', alpha=0.8)
bars3 = ax.bar(x + width, f1, width, label='F1-Score', color='#22c55e', alpha=0.8)

ax.set_xlabel('Classe', fontsize=12)
ax.set_ylabel('Score', fontsize=12)
ax.set_title('Métricas por Classe - Classificação de Texturas', fontsize=14,
fontweight='bold')
ax.set_xticks(x)
ax.set_xticklabels(classes_textura)
ax.legend(loc='upper right')
ax.set_ylim(0, 1.35)
ax.grid(axis='y', alpha=0.3)

for bars in [bars1, bars2, bars3]:
    for bar in bars:
        height = bar.get_height()
        ax.text(bar.get_x() + bar.get_width()/2., height + 0.02,
                f'{height:.2f}', ha='center', va='bottom', fontsize=9)

plt.tight_layout()
plt.show()

print("Interpretation of the metrics:")
print("- Precision: among the samples classified as belonging to the class,",
      "how many were correct?")
print("- Recall: among the samples that actually belong to the class, how many",
      "were identified?")
print("- F1-Score: harmonic mean between precision and recall.")
print("- Support: number of actual samples of each class present in the test set.")
print("\nSupport does not measure performance; it only informs how many examples of each",
      "class were \nused in the evaluation.")
```

```
=== DETAILED CLASSIFICATION REPORT ===
Class          Precision       Recall   F1-score    Support
granular            0.67         0.78       0.72         18
listrada            0.55         0.61       0.58         18
manchada            1.00         0.72       0.84         18
```

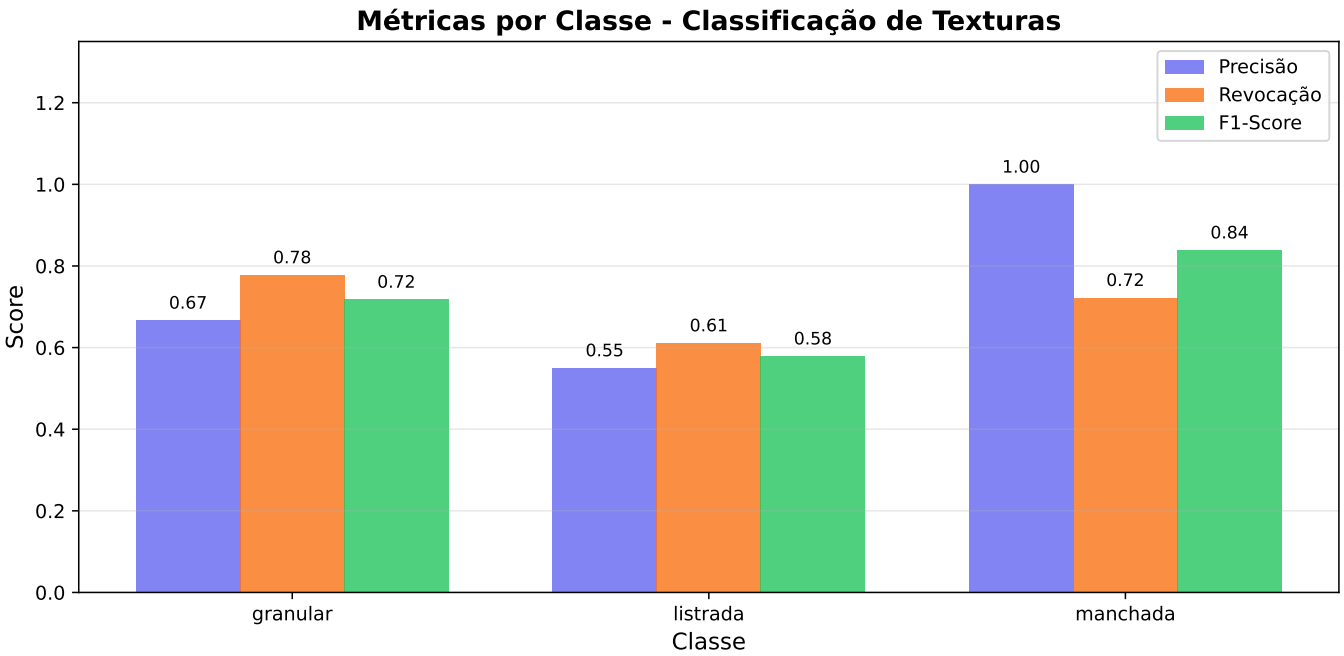


Figure 7.20: Detailed evaluation metrics for the k-NN classifier with LBP descriptors

```
Interpretation of the metrics:
- Precision: among the samples classified as belonging to the class, how many were correct?
- Recall: among the samples that actually belong to the class, how many were identified?
- F1-Score: harmonic mean between precision and recall.
- Support: number of actual samples of each class present in the test set.

Support does not measure performance; it only informs how many examples of each class were
used in the evaluation.
```

7.14 Limitations of Handcrafted Descriptors

The experiments in this chapter show that classical descriptors can be quite effective in classification tasks, but they also present important limitations:

- **Specificity:** each descriptor was developed to represent a particular type of information, such as color, texture, or shape. Thus, a descriptor suitable for one task may not be the most appropriate for another.
- **Hyperparameter dependence:** the performance of descriptors such as LBP and HOG depends on the choice of parameters, such as neighborhood radius, number of sampled points, cell size, and number of orientations, which need to be adjusted according to the application.
- **Limited representation:** color, texture, and gradient descriptors capture low-level properties of the image, but they do not directly represent more complex semantic concepts, such as objects or scenes.
- **Curse of dimensionality:** overly extensive descriptors can reduce the effectiveness of distance-based classifiers, such as k-NN.

These limitations motivate the evolution of the techniques studied in the next chapters. **Chapter 8** presents classical methods for detection and matching of features in images, while **Chapter 9** introduces **Convolutional Neural Networks**, capable of automatically learning appropriate representations for each task from the data.

7.15 Summary

This chapter presented the fundamentals of pattern recognition applied to images. The main concepts studied were:

- ***Pipeline* of pattern recognition:** acquisition, preprocessing, descriptor extraction, classification, and evaluation.
- **Classical descriptors:** color descriptors, LBP for texture, and HOG for shape, used to represent different characteristics of images.
- **Feature normalization:** standardization (*Z-score*) to prevent attributes of greater magnitude from dominating distance calculations.
- ***k*-NN classifier:** classification based on the k nearest neighbors in feature space.
- **Choice of parameter k:** influence of the value of k on classifier performance and the use of cross-validation for its selection.
- **Classifier evaluation:** accuracy, confusion matrix, precision, recall, and F1-score as complementary performance metrics.
- **Limitations of handcrafted descriptors:** specificity, dependence on hyperparameters, and difficulty in representing high-level information.

The concepts were illustrated through experiments with the public dataset `load_digits`, synthetic textures generated for educational purposes, and simulated fruit descriptor data.

7.16 🤖 Using Gemini Notebook as a Tutor

In this edition, **Gemini Notebook** is presented as a study support tool. The system uses exclusively the documents provided by the author as a source of knowledge, allowing you to explore the chapter's concepts through questions, summaries, and explanations related to the studied material.

🎓 Study with the Intelligent Tutor

🚀 ACCESS Gemini Notebook: CHAPTER 07

🌐 **Language and Programming Language**

The project for this chapter in Gemini Notebook was built solely with the text in **Portuguese** and the code examples in **Python**. If you are studying from the English or French edition, or following the C++ track, the tutor's answers may not correspond exactly to the version you are reading.

⚠️ **Notice about AI-Generated Content**

Answers provided by Gemini Notebook may contain inaccuracies or omissions. When necessary, confirm the information using the material from this chapter and other reliable academic sources. Running the practical examples presented throughout the text remains the best way to consolidate the concepts studied.

7.17 Exercise List

The following exercises explore and extend the concepts presented in this chapter through adaptations of the implemented algorithms, experimental analyses, and comparisons between different approaches.

1. **(10%)** Implement a color descriptor (RGB or HSV histogram, with at least 16 *bins* per channel) for the three classes of simulated fruits in Figure 7.1. Train a k-NN classifier with this descriptor, compare its accuracy with that obtained by the LBP and HOG descriptors (Figure 7.9), and discuss in which situations color information is more discriminative.

2. **(15%)** Investigate the effect of feature normalization (*Z-score*) on the performance of k-NN in a heterogeneous attribute space, combining color, LBP, and HOG descriptors into a single vector. Compare the results obtained with and without normalization for at least three values of k.

3. **(15%)** Reproduce the *overfitting* and *underfitting* analysis from Figure 7.17 by varying the training

set size (e.g., 20%, 50%, and 80% of the `load_digits` dataset). Discuss how the number of examples influences the choice of the k value.

4. **(15%)** Extend Practical Project 2 by adding a fourth synthetic texture class. Evaluate precision, recall, and F1-score for each class, following the pattern in Figure 7.20, and analyze the impact of the new class on the confusion matrix.

5. **(15%)** Manually implement the k-NN classifier, without using `sklearn`:

   `sklearn.neighbors.KNeighborsClassifier`,

   completing the `knn_passo_a_passo` function presented in the chapter. Compare the accuracy and execution time of the manual implementation with the `scikit-learn` implementation on datasets of increasing sizes and relate the results to the curse of dimensionality.

6. **(15%)** Investigate the influence of the `orientations`, `pixels_per_cell`, and `cells_per_block` parameters of the HOG descriptor on the `load_digits` dataset. Evaluate at least four parameter combinations and discuss the trade-off between descriptor dimensionality and classifier performance.

7. **(15%)** Evaluate the influence of the parameters P (number of neighbors) and R (radius) of the LBP descriptor on the classification of the synthetic textures from Practical Project 2, considering $P \in \{4, 8, 16\}$ and $R \in \{1, 2, 3\}$. Analyze how these parameters affect the discriminative capability of the descriptor.

8. **(Bonus – 10%)** Manually implement k-fold cross-validation for the k-NN classifier on the `load_digits` dataset, without using `cross_val_score`, and compare the results with those obtained by the `scikit-learn` implementation presented in Figure 7.16.

Chapter References

The theoretical foundation and experiments presented in this chapter are based on the following references:

- Gonzalez (2018), for the fundamentals of statistical texture descriptors and the preprocessing operations applied to feature extraction.
- Szeliski (2022), for the presentation of the classical pattern recognition pipeline, descriptor extraction, and classifier evaluation in Computer Vision.
- Duda (2001), for the theoretical foundations of pattern recognition, the k-NN classifier, and the bias-variance trade-off.
- Cover (1967), for the original formulation of the k nearest neighbors algorithm.
- Ojala (2002), for the formulation of the *Local Binary Patterns* (LBP) descriptor and its uniform variant, used in this chapter.
- Dalal (2005), for the formulation of the *Histogram of Oriented Gradients* (HOG) descriptor, employed in shape and contour representation.
- Pedregosa (2011), for the implementation of the k-NN classifier, evaluation metrics, and cross-validation in the `scikit-learn` library.
- Quilici-gonzalez (2014), for the didactic presentation of traditional Pattern Recognition classifiers, such as Decision Trees, Classification Rules, and Support Vector Machines (SVM), complementary to the k-NN classifier explored in this chapter.
- Quilici-gonzalez (2026), for the update and expansion of these contents in its 2nd edition, currently in production.

Run Colab Open GitHub

7.18 💻 Practical Part with Programming Exercises

This list of programming exercises (EP) consolidates the theoretical formulations presented throughout Chapter 7 — Image Classification and Pattern Recognition — through an applied practical track. Unlike the direct pixel manipulation of the previous chapters, the EPs in this chapter work with the **intermediate quantities** of a real pattern recognition *pipeline* — feature vectors, distances, predicted and actual labels, local binary patterns, and orientation histograms — allowing each step of the reasoning to be manually validated without relying on external machine learning libraries.

The sequencing of the exercises reproduces the conceptual flow of the chapter: it begins with the manual implementation of the **k-NN** classifier decision rule over a small feature space; then, from the perspective of feature normalization, the classifier implemented in the first exercise of the list is revisited; it proceeds to the calculation of the **evaluation** metrics (confusion matrix, precision, and recall) from predicted and actual labels; it continues with the manual coding of the **LBP** texture descriptor from a 3×3 neighborhood; it delves into the computation of the orientation histogram of the **HOG** descriptor for a single cell; it then advances to the integration of **descriptor extraction**, **k-NN classification**, and **multi-class evaluation** into a complete texture recognition *pipeline*; and it concludes with the application of this same *pipeline* to a **real image** (PGM format), in which the LBP descriptor is computed directly on the pixels of a texture mosaic.

🎯 Objective of this Notebook

This notebook allows you to develop, validate, organize, and test solutions for **Programming Exercises (EPs)** in interactive environments, such as Colab, using the same test cases as Moodle, and copying them there only when registering the official grade.

Download

Download `morph.py` and `testsuite.py` by running the cell below:

```
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup(testsuite=True)
from morph import mm
from testsuite import TestSuite
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Running the Tests

To evaluate the tests, run `TestSuite("EP07_01.extension").run()` in a new cell, replacing the extension with that of the language used (`.py`, `.java`, `.c`, `.cpp`, `.js`, or `.r`). The system downloads the test cases from GitHub, runs the program, and calculates the grade automatically.

To test Python code directly, without saving a file, use `run_code(code)` passing the code as a *string* in a `code` variable:

```
code = """
# ... your code here ...
"""
TestSuite("EP07_01").run_code(code)
```

Summary of the `morph.py` Methods (Ch. 7)

The `morph.py` library provides two versions for most algorithms: a **pedagogical** one (methods ending in `0`), implemented step by step in NumPy, and a **classical** one, based on the `scikit-learn` and `scikit-image` libraries. The pedagogical implementations are used in the **Programming Exercises (PEs)**, as they do not depend on external libraries and run within the memory limits of the Moodle **VPL** environment. The classical versions, in turn, are more efficient and suitable for experiments in environments such as Colab and Jupyter Notebook, but they usually **cannot be used in the Moodle PEs**, as the `scikit-learn` library exceeds the memory available in the VPL.

1. **Data reading (`readClasses`, `readDataset`, `readTrain`, `readTest`)**
   They standardize the input of the training and test sets, returning the feature matrices (X) and the label vectors (y).
2. **Classification (`knn0` / `knn`)**
   They implement the **k-nearest neighbors (k-NN)** algorithm for binary and multiclass classification, using Euclidean or Manhattan distance.
3. **Normalization (`zscore0` / `zscore`)**
   They apply *z-score* normalization to the attributes, reducing scale differences before classification.
4. **Evaluation (`confusion0` / `confusion`)**
   They compute the confusion matrix and metrics such as accuracy, precision, and recall, for both binary and multiclass problems.
5. **Texture descriptor (`lbp0` / `lbp`)**
   They compute the ***Local Binary Pattern* (LBP)**, allowing one to obtain the LBP map, the code of a single pixel, or the histogram of an image region.
6. **Shape descriptor (`hog0` / `hog`)**
   They compute the ***Histogram of Oriented Gradients* (HOG)**, producing histograms of gradient orientations to represent shape and contour information.

7.18.1 EP07_01 🟢 Step-by-Step k-NN Classifier

The `KNeighborsClassifier` from `scikit-learn`, used throughout the chapter, hides behind a single call (`.fit` / `.predict`) a quite simple decision rule: for each new observation, compute the distance to all training examples, select the k closest ones, and vote by the majority class among them.

Before relying on the library, you have been tasked with implementing this rule from scratch, for a two-dimensional feature space, exactly as the chapter's interactive decision boundary simulator does internally with each user click.

7.18.1.1 📋 Implementation Guidelines

1. **Quantity and parameter:** Read the integer N (number of training examples) and the odd integer k (number of neighbors).
2. **Training examples:** For each of the N examples, read three values: the x and y coordinates (real numbers) and the label r (integer, 0 or 1).
3. **Queries:** Read the integer Q (number of query points) and then the coordinates x_q, y_q (real numbers) of each query.
4. **Distance:** For each query, compute the Euclidean distance to **all** training examples:

$$d(x_q, x_i) = \sqrt{(x_q - x_i)^2 + (y_q - y_i)^2}.$$

5. **Neighbor selection:** Sort the examples by increasing distance and select the first k. In case of a **distance tie** at the boundary of the k-th neighbor, break the tie by the example read **first** in the input (stable reading order).

6. **Majority voting:** Count the votes for each class among the k selected neighbors. If there is a **tie in the voting** (only possible when k is even, which should not occur per the guideline in item 1, but handle defensively), assign the class of the closest neighbor among the tied classes.
7. **Output:** For each query, in input order, print the predicted class. At the end, print the total number of queries classified as class `1`.

7.18.1.2 📌 Computational Constraints

- **Fixed metric:** use exclusively the Euclidean distance (not the *squared distance*) for the sorting, although the comparison result is the same.
- k **always odd:** the input guarantees k odd and $k \leq N$; still, implement the tie-breaking from item 6 for robustness.
- **Stability:** when sorting by distance, preserve the relative order of examples with the same distance (stable sorting).

7.18.1.3 🧠 Theoretical Foundation

Element	Role in k-NN
Feature space	Set of all possible vectors (x, y)
Euclidean distance	Measure of similarity between observations
Small k	Irregular boundary, high variance
Large k	Smooth boundary, high bias
Majority voting	Decision rule $\hat{y} = \text{mode}\{y_i : x_i \in N_k(x)\}$

7.18.1.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integers N and k, separated by spaces.
- Next N lines: three values per line — x, y (real numbers) and r (integer $\in \{0, 1\}$), separated by spaces.
- Next line: integer Q.
- Next Q lines: two values per line — x_q, y_q (real numbers), separated by spaces.

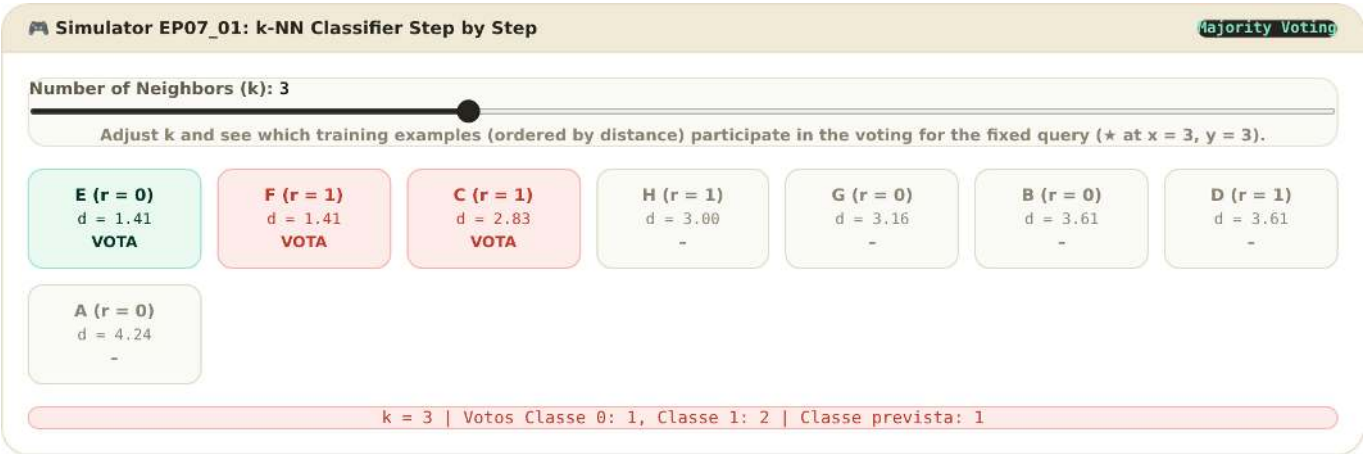
Output:

- Q lines, each with the predicted class (`0` or `1`) for the respective query, in input order.
- Last line: `Total class 1: X`.

7.18.1.5 📌 Examples

Input	Output	Observation
4 3 0 0 0 1 0 0 5 5 1 6 5 1 1 1 1	0 Total class 1: 0	Query close to the class 0 cluster.

Input	Output	Observation
4 1 0 0 0 1 0 0 5 5 1 6 5 1 2 0.9 0.1 5.5 5.1	0 1 Total class 1: 1	With $k = 1$, each query inherits the class of its closest neighbor.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap07/sim-ep0701.html

Figure 7.21: EP07_01 Simulator: Step-by-Step k-NN Classifier

```
%%writefile EP07_01.py
# Python code
```

```
Overwriting EP07_01.py
```

```
TestSuite("EP07_01.py").run()
```

```
<IPython.core.display.HTML object>
```

7.18.2 EP07_02 🟡 *Z-score* Normalization and Robustness of k-NN to Distinct Scales

This exercise revisits the classifier implemented in **EP07_01**, this time under the perspective discussed in the section *The Impact of Scale and Feature Normalization* of the chapter: k-NN decides based on the distance between vectors, so that a feature measured on a much larger scale than the others tends to **dominate** the distance calculation, even when it is not the most relevant for separating the classes.

An inspection system records, for each part, its **area** (in pixels, possibly reaching hundreds or thousands) and its **circularity** (always between 0 and 1). You have been tasked with classifying new parts by k-NN in two ways — with and without the *Z-score* standardization presented in the chapter — and reporting in which cases the two approaches **diverge**.

7.18.2.1 📋 Implementation Guidelines

1. **Quantity and parameter:** Read the integer N (number of training examples) and the odd integer k.
2. **Training examples:** For each of the N examples, read three values: the area x_1 (real), the circularity x_2 (real), and the label r (integer, 0 or 1).
3. **Queries:** Read the integer Q and then the coordinates x_1, x_2 of each query.

4. **Classification without normalization:** For each query, classify it by k-NN directly on (x_1, x_2), with Euclidean distance and the same tie-breaking rules as in EP07_01 (reading order for tied distances; nearest neighbor among tied classes in the majority vote).
5. **Normalization parameters:** Compute the mean μ_j and the **population** standard deviation σ_j (division by N, not $N-1$ — the same convention adopted by the `StandardScaler` class) of each feature $j \in \{1, 2\}$, **exclusively on the training set**.
6. **Standardization:** Transform each training and query feature by

$$z_j = \frac{x_j - \mu_j}{\sigma_j}.$$

   If $\sigma_j = 0$ (constant feature in the training set), set $z_j = 0$ for all samples of that feature, avoiding division by zero.
7. **Classification with normalization:** Repeat the k-NN classification of item 4, now on the standardized vectors (z_1, z_2), with the same tie-breaking rules.
8. **Output:** For each query, in input order, print the two predicted classes. At the end, print the number of queries in which the two classifications **diverge**.

7.18.2.2 📌 Computational Constraints

- **Fit on training data only:** μ_j and σ_j are computed solely from the training set and reapplied to the queries — never recalculated from them. This practice avoids **data leakage**, mentioned in the normalization section of the chapter.
- **Population standard deviation:** use $\sigma_j = \sqrt{\frac{1}{N}\sum_i (x_{i,j} - \mu_j)^2}$, not the sample version (division by $N-1$).
- **Constant feature:** treat $\sigma_j = 0$ as a special case (item 6); no division-by-zero error should occur.
- **Tie-breaking rules:** reuse exactly the conventions from EP07_01, both in selecting the k neighbors and in the majority vote.

7.18.2.3 🧠 Theoretical Foundation

Element	Role
Z-score standardization	Rescales each feature to mean 0 and standard deviation 1, making heterogeneous scales comparable
Fit on training data only	Ensures that evaluation on queries reflects only what the model learned from the training data
Euclidean distance without normalization	Dominated by the feature with the largest magnitude — here, the area
Divergent prediction	Highlights that the scale of features, not just the algorithm or the data, can determine the k-NN decision boundary

This exercise reinforces, in a controlled manner, the reason why `StandardScaler` is applied before k-NN throughout the chapter: without this step, circularity features — even when highly discriminative — can be practically ignored by the classifier in the presence of an area feature with a magnitude hundreds of times larger.

7.18.2.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integers N and k, separated by a space.
- Next N lines: three values per line — x_1, x_2 (real) and r (integer $\in \{0, 1\}$), separated by a space.
- Next line: integer Q.

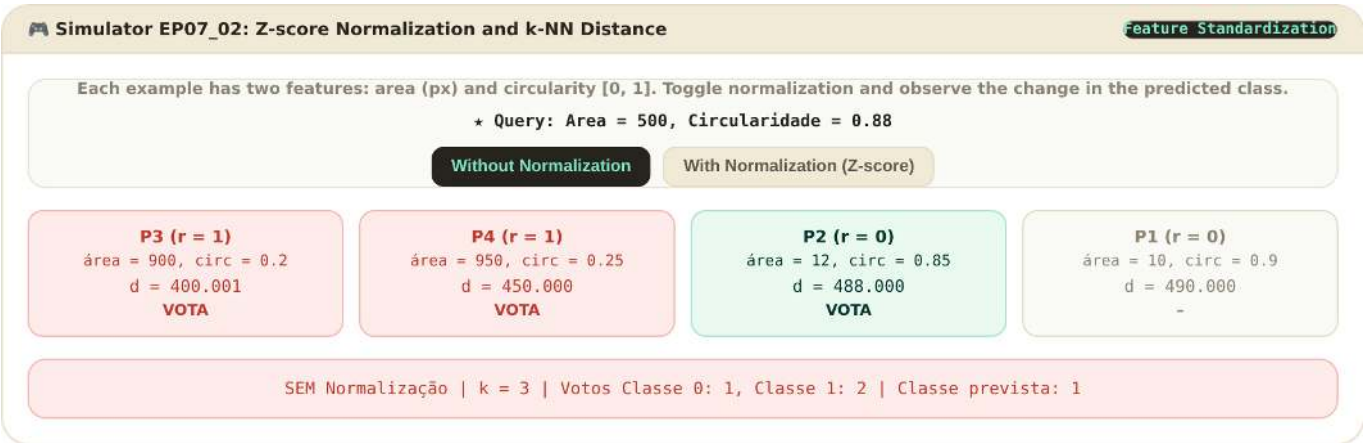
- Next Q lines: two values per line — x_1, x_2 (real) of the query, separated by a space.

Output:

- Q lines, in the format `SemNorm=<0|1> ComNorm=<0|1>`, in the input order of the queries.
- Last line: `Divergiu: <int>`.

7.18.2.5 📌 Examples

Input	Output	Observation
4 3 10 0.9 0 12 0.85 0 900 0.2 1 950 0.25 1 1 500 0.88	SemNorm=1 ComNorm=0 Divergiu: 1	Without normalization, the area (scale of hundreds) dominates the distance, and the query is classified as class `1`. After standardization, the circularity — much closer to the class `0` samples — becomes comparably weighted, and the prediction changes to `0`.
2 1 0 0.5 0 100 0.5 1 1 60 0.5	SemNorm=1 ComNorm=1 Divergiu: 0	The circularity is constant in the training set ($\sigma_2 = 0$); by the rule of item 6, $z_2 = 0$ for all samples, and the classification depends only on the area in both cases.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap07/sim-ep0702.html

Figure 7.22: EP07_02 Simulator: Effect of *Z-score* Normalization on k-NN Distance

```
%%writefile EP07_02.py
# Python code
```

```
Overwriting EP07_02.py
```

```
TestSuite("EP07_02.py").run()
```

```
<IPython.core.display.HTML object>
```

7.18.3 EP07_03 🟡 Evaluation via Confusion Matrix

A binary weld quality classifier was trained and tested on a production line. For each inspected part, the system recorded the **actual** label (obtained by an expert) and the **predicted** label from the classifier, where `1` represents "defective" and `0` represents "conforming."

Quality management wants to know not only the system's accuracy but also its **precision** (when the system flags a defect, how often is it correct?) and its **recall** (of all truly defective parts, how many did the system manage to identify?)—the distinction discussed in the classifier evaluation section of the chapter.

7.18.3.1 📋 Implementation Guidelines

1. **Quantity:** Read the integer N (number of inspected parts).
2. **Data for each part:** For each of the N parts, read two integers—the actual label y and the predicted label $\hat{y}$ (both $\in \{0, 1\}$).
3. **Confusion matrix:** Considering class `1` (defective) as **positive**, count:
   - VP (True Positive): $y = 1$ and $\hat{y} = 1$;
   - FP (False Positive): $y = 0$ and $\hat{y} = 1$;
   - FN (False Negative): $y = 1$ and $\hat{y} = 0$;
   - VN (True Negative): $y = 0$ and $\hat{y} = 0$.
4. **Metrics:** Calculate

$$\text{Accuracy} = \frac{VP + VN}{N}, \quad \text{Precision} = \frac{VP}{VP + FP}, \quad \text{Recall} = \frac{VP}{VP + FN}.$$

5. **Degenerate cases:** If $VP + FP = 0$ (no positive predictions), print `Precisao: indefinida`. If $VP + FN = 0$ (no actual positive cases), print `Revocacao: indefinida`.
6. **Rounding:** All numerical metrics must be rounded to 4 decimal places (*round half away from zero*) only for display.

7.18.3.2 📌 Computational Constraints

- **Fixed positive class convention:** class `1` is always the positive class in this exercise, regardless of its relative frequency.
- **Division by zero protection:** implement the degenerate cases from item 5 before performing the division.
- **Output order:** follow exactly the order specified in the output section, even in degenerate cases.

7.18.3.3 🧠 Theoretical Foundation

Metric	Question It Answers	Sensitive to Imbalance?
Accuracy	What fraction of parts was classified correctly?	Yes—can mask errors in the minority class
Precision	Of the parts flagged as defective, how many truly are?	Penalizes false positives
Recall	Of the truly defective parts, how many were detected?	Penalizes false negatives

In an industrial context, **low recall** is often more severe than **low precision**: letting a defective part pass (false negative) tends to be costlier than manually inspecting a good part flagged by mistake (false positive).

7.18.3.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer N.
- Next N lines: two integers per line—y and $\hat{y}$, separated by spaces.

Output (in this exact order):

```
VP=<int> FP=<int> FN=<int> VN=<int>
Acuracia: <value or undefined metric>
Precisao: <value or undefined>
Revocacao: <value or undefined>
```

7.18.3.5 📌 Examples

Input	Output	Observation
4 1 1 0 1 1 0 0 0	VP=1 FP=1 FN=1 VN=1 Acuracia: 0.5000 Precisao: 0.5000 Revocacao: 0.5000	One error of each type.
3 0 0 0 0 0 0	VP=0 FP=0 FN=0 VN=3 Acuracia: 1.0000 Precisao: indefinida Revocacao: indefinida	No actual or predicted positive cases.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap07/sim-ep0703.html

Figure 7.23: Simulator EP07_03: Precision x Recall

```
%%writefile EP07_03.py
# Python code
```

```
Overwriting EP07_03.py
```

```
TestSuite("EP07_03.py").run()
```

```
<IPython.core.display.HTML object>
```

7.18.4 EP07_04 🟠 Manual Encoding of the LBP Descriptor

The `local_binary_pattern` function from `scikit-image`, used in the texture classification project, automatically computes the LBP code for each pixel of an image. Before using it as a black box, you have been tasked with manually implementing the computation of the classic LBP code ($P = 8$, $R = 1$) for the central pixel of a 3×3 neighborhood, exactly as defined in the equation in the chapter.

In addition to the code, the texture inspection system also needs to know whether that pattern is **uniform** — a pattern is uniform when the number of transitions ($0 \to 1$ or $1 \to 0$) when traversing the 8 bits **circularly** (returning from the last bit to the first) is **at most 2**, a property exploited by the *uniform* variant of the LBP mentioned in the chapter.

7.18.4.1 📋 Implementation Guidelines

1. **Quantity:** Read the integer T (number of neighborhoods to process).
2. **Data for each neighborhood:** For each of the T neighborhoods, read a 3×3 matrix of integers (intensities), provided in 3 lines of 3 values each. The central pixel is at position `[1][1]`.
3. **Order of neighbors:** Traverse the 8 neighbors in **clockwise** order, starting at the top-left corner, in the following sequence of positions `[row][column]`: `[0][0]`, `[0][1]`, `[0][2]`, `[1][2]`, `[2][2]`, `[2][1]`, `[2][0]`, `[1][0]`. This corresponds to the index $p = 0, 1, \dots, 7$ in the LBP equation.
4. **Threshold function:** For each neighbor p with intensity g_p and center g_c, compute $s(g_p - g_c)$, which is `1` if $g_p \geq g_c$ and `0` otherwise.
5. **LBP code:** Compute
$$\text{LBP} = \sum_{p=0}^{7} s(g_p - g_c)\, 2^p.$$
6. **Transitions:** Considering the circular bit sequence $s_0, s_1, \dots, s_7$ (in the order from item 3), count how many consecutive pairs **adjacent in the circular sequence** (including the pair s_7, s_0) differ from each other.
7. **Classification:** If the number of transitions is ≤ 2, classify as `UNIFORME`; otherwise, classify as `NAO_UNIFORME`.
8. **Output:** For each neighborhood, in input order, print the LBP code (decimal integer, 0–255), the number of transitions, and the classification.

7.18.4.2 📌 Computational Constraints

- **Fixed neighbor order:** The order from item 3 is mandatory — reversing it produces a numerically different code, even though it represents the same visual pattern.
- **Non-strict comparison:** $s(z) = 1$ when $z \geq 0$ (the chapter itself defines equality as included in the `1` case).
- **Circular counting:** Do not forget the pair that closes the cycle (s_7 with s_0); ignoring this pair is a common mistake that incorrectly classifies uniform patterns.

7.18.4.3 🧠 Theoretical Foundation

Pattern (bits $s_0 \dots s_7$)	Transitions	Interpretation
`00000000` or `11111111`	0	Homogeneous region (light or dark patch)
`00001111`	2	Simple edge between two regions
`01010101`	8	Alternating contrast texture — non-uniform

Uniform patterns are concentrated in smooth texture regions or simple edges; non-uniform patterns tend to correspond to high-frequency noise. For this reason, the *uniform* LBP histogram, used in the texture classification project, groups all non-uniform patterns into a single bin, reducing the dimensionality of the descriptor.

7.18.4.4 📦 Input and Output Specification (VPL)

Input:

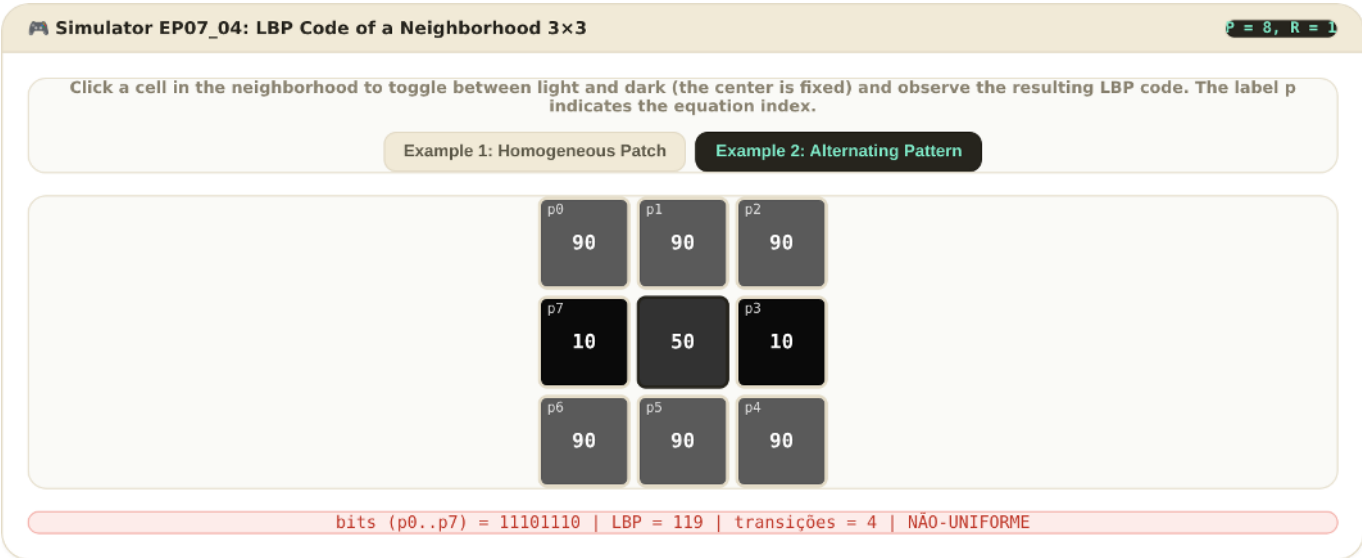
- Line 1: Integer T.
- For each neighborhood: 3 lines with 3 integers each (3×3 matrix).

Output:

- T lines, in the format `LBP=<int> transicoes=<int> <UNIFORME|NAO_UNIFORME>`.

7.18.4.5 📌 Examples

Input	Output	Observation
1 10 10 10 10 50 10 10 10 10	LBP=0 transicoes=0 UNIFORME	Center is the brightest; all neighbors generate bit 0.
1 90 90 90 10 50 10 90 90 90	LBP=119 transicoes=4 NAO_UNIFORME	Light and dark neighbors alternate in the neighborhood.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap07/sim-ep0704.html

Figure 7.24: EP07_04 Simulator: LBP Code of a 3x3 Neighborhood

```
%%writefile EP07_04.py
# Python code
```

```
Overwriting EP07_04.py
```

```
TestSuite("EP07_04.py").run()
```

```
<IPython.core.display.HTML object>
```

7.18.5 EP07_05 🔴 Histogram of Orientations for an HOG Cell

The `hog` function from `scikit-image`, used in the digit classification project, divides the image into small **cells** and, for each one, builds a histogram of gradient orientations weighted by magnitude — exactly the central step described in the section on the HOG descriptor in the chapter.

You have been tasked with implementing this computation for a single cell, using the magnitude and gradient orientation values **already computed** for each pixel in the cell (dispensing with the computation of partial derivatives).

7.18.5.1 📋 Implementation Guidelines

1. **Dimensions:** Read the integers n (the cell has $n \times n$ pixels) and B (number of histogram bins).

2. **Magnitudes:** Read n lines with n real values each, representing $|\nabla f(x, y)|$ for each pixel in the cell.
3. **Orientations:** Read n more lines with n real values each, representing $\theta(x, y)$ in **degrees**, already converted to the **unsigned** interval $[0^\circ, 180^\circ)$, as conventionally used by HOG.
4. **Bins:** The B bins cover $[0^\circ, 180^\circ)$ in equal bands of width $180/B$ degrees. A pixel with orientation θ belongs to bin $\lfloor \theta/(180/B) \rfloor$; if this index equals B (possible only when θ is exactly 180°, which should not occur according to the guideline in item 3), use bin $B - 1$.
5. **Raw histogram:** For each pixel, accumulate its **magnitude** (not its count) in the corresponding bin:

$$H[b] = \sum_{(x,y)\,:\,\mathrm{bin}(\theta(x,y))=b} |\nabla f(x, y)|.$$

6. **L2 normalization:** After constructing H, normalize it to obtain $\hat{H}$:

$$\hat{H}[b] = \frac{H[b]}{\sqrt{\sum_{j=0}^{B-1} H[j]^2 + \epsilon}}, \qquad \epsilon = 10^{-6}.$$

7. **Output:** Print the raw histogram H (rounded to 2 decimal places) on one line, followed by the normalized histogram $\hat{H}$ (rounded to 4 decimal places) on another line, both with the B values separated by spaces, in bin order.

7.18.5.2 📌 Computational Constraints

- ***Unsigned binning*:** the orientation interval is $[0, 180)$, not $[0, 360)$ — gradients in opposite directions (differing by 180°) contribute to the **same** bin, the standard HOG convention for object detection.
- **Accumulation by magnitude, not by count:** the histogram weights each pixel by its gradient magnitude, rather than simply counting how many pixels fall into each bin.
- **Stabilization constant:** the $\epsilon = 10^{-6}$ in the normalization denominator avoids division by zero when the cell is completely homogeneous (all magnitudes zero).

7.18.5.3 📐 Where the Input Matrices Come From

Before this assignment, each pixel (x, y) of the image undergoes:

$$G_x = f(x + 1, y) - f(x - 1, y), \qquad G_y = f(x, y + 1) - f(x, y - 1)$$

$$|\nabla f| = \sqrt{G_x^2 + G_y^2}, \qquad \theta_{\text{signed}} = \mathrm{atan2}(G_y, G_x)$$

Since HOG disregards contrast polarity, the angle is folded into the unsigned interval:

$$\theta = \theta_{\text{signed}} \bmod 180^\circ$$

Repeating this for all pixels in an $n \times n$ cell yields the two input matrices for this exercise: **magnitudes** $|\nabla f|$ and **orientations** $\theta \in [0^\circ, 180^\circ)$.

7.18.5.4 🧠 Theoretical Background

Step	Role
Gradient magnitude	Weights each pixel's contribution — strong edges weigh more than weak noise
Unsigned orientation	Makes the descriptor invariant to contrast polarity (light→dark vs. dark→light)

Step	Role
Per-cell histogram	Summarizes the local edge distribution into a compact vector
L2 normalization	Reduces the descriptor's sensitivity to global illumination and contrast variations

The concatenation of the normalized histograms from all cells in the image — not implemented in this exercise — forms the complete HOG feature vector, used as input to the k-NN classifier in the chapter's project.

7.18.5.5 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integers n and B.
- Next n lines: n real magnitudes each.
- Next n lines: n real orientations (degrees, $[0, 180)$) each.

Output:

- Line 1: the B values of the raw histogram, rounded to 2 decimal places.
- Line 2: the B values of the normalized histogram, rounded to 4 decimal places.

7.18.5.6 📌 Examples

Input	Output	Observation
2 2 1.0 2.0 3.0 4.0 10 100 170 20	5.00 5.00 0.7071 0.7071	Bin width 90°: $[0, 90)$ and $[90, 180)$; magnitudes 1 and 4 fall into bin 0, 2 and 3 into bin 1.
2 4 0.0 0.0 0.0 0.0 0 0 0 0	0.00 0.00 0.00 0.00 0.0000 0.0000 0.0000 0.0000	Homogeneous cell: ϵ avoids division by zero.

```
%%writefile EP07_05.py
# Python code
```

```
Overwriting EP07_05.py
```

```
TestSuite("EP07_05.py").run()
```

```
<IPython.core.display.HTML object>
```

7.18.6 EP07_06 🟣 Complete *Pipeline*: Descriptors + k-NN + Multi-Class Evaluation

This exercise integrates the three central stages of the chapter into a single *pipeline*, reproducing in miniature the **Practical Project 2** (classification of synthetic textures by LBP): a set of histograms of descriptors **already extracted** (as if they were LBP histograms) is used to train a k-NN classifier, which in turn is evaluated on an independent test set using a multi-class confusion matrix.

🎮 **Simulator EP07_05: Orientation Histogram of a Cell** — fixed 3×3 cell

Adjust B and watch how the **orientation matrix** (independent of the magnitudes one) is mapped to the bins via `bin = floor(θ / (180/B))`, and how the magnitudes are summed into each bin.

Number of bins (B) 2

📄 **Input (exactly as the program reads from stdin)**

```
3 2
1.0 2.0 0.5
3.0 4.0 1.5
0.8 2.5 3.2
10 100 45
170 20 95
60 150 5
```

🔢 **Magnitude matrix |∇f|**

1.0	2.0	0.5
3.0	4.0	1.5
0.8	2.5	3.2

📐 **Orientation matrix θ (degrees) — colored by bin**

10° +90.0=0.11 ⌊·⌋=bin 0	100° +90.0=1.11 ⌊·⌋=bin 1	45° +90.0=0.50 ⌊·⌋=bin 0
170° +90.0=1.89 ⌊·⌋=bin 1	20° +90.0=0.22 ⌊·⌋=bin 0	95° +90.0=1.06 ⌊·⌋=bin 1
60° +90.0=0.67 ⌊·⌋=bin 0	150° +90.0=1.67 ⌊·⌋=bin 1	5° +90.0=0.06 ⌊·⌋=bin 0

📏 **Where each θ falls on the ruler [0°, 180°) — bin = floor(θ / width)**

(2,2)(0,0) (1,1) (0,2) (2,0) (1,2)(0,1) (2,1) (1,0)

0° 90° 180°

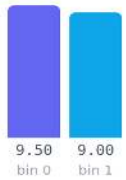
📊 **Range of each bin (width = 180° / B)**

bin 0: [0.0°, 90.0°) bin 1: [90.0°, 180.0°)

🧩 **Each pixel: magnitude + orientation → bin**

mag=1.0 θ=10° → bin 0	mag=2.0 θ=100° → bin 1	mag=0.5 θ=45° → bin 0
mag=3.0 θ=170° → bin 1	mag=4.0 θ=20° → bin 0	mag=1.5 θ=95° → bin 1
mag=0.8 θ=60° → bin 0	mag=2.5 θ=150° → bin 1	mag=3.2 θ=5° → bin 0

Raw histogram (H) | Normalized histogram (Ĥ)



🧮 **Step-by-step calculation (floor of the division + summing magnitudes per bin)**

H[0] = 1.00 (θ=10°→⌊0.11⌋=0) + 0.50 (θ=45°→⌊0.50⌋=0) + 4.00 (θ=20°→⌊0.22⌋=0) + 0.80 (θ=60°→⌊0.67⌋=0) + 3.20 (θ=5°→⌊0.06⌋=0) = **9.50**

H[1] = 2.00 (θ=100°→⌊1.11⌋=1) + 3.00 (θ=170°→⌊1.89⌋=1) + 1.50 (θ=95°→⌊1.06⌋=1) + 2.50 (θ=150°→⌊1.67⌋=1) = **9.00**

H=[9.50, 9.00] | Ĥ=[0.7260, 0.6877]

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap07/sim-ep0705.html

Figure 7.25: EP07_05 Simulator: HOG Histogram of a Cell (mapping angles to bins)

Unlike EP07_01, here the feature space has arbitrary dimension H (the histogram size), there are more than two classes, and the distance metric is an input parameter — making it possible to reproduce the metric comparison experiment discussed in the chapter.

7.18.6.1 📋 Implementation Guidelines

1. **Classes:** Read the integer C (number of classes) followed by C class names (*strings* without spaces), in the order in which they should appear in the confusion matrix.
2. **Configuration:** Read the integer H (histogram dimension), the *string* M (metric: `euclidiana` or `manhattan`) and the odd integer k.
3. **Training:** Read the integer N and then N lines, each containing the class name followed by H real values (the descriptor histogram).
4. **Testing:** Read the integer Q and then Q lines, each containing the **actual** class name followed by H real values (the descriptor histogram of the test sample).
5. **Distance:** For each test sample, calculate the distance to each training example using the metric M:

$$d_{\text{euclidiana}}(u,v) = \sqrt{\sum_{j=1}^{H}(u_j - v_j)^2}, \qquad d_{\text{manhattan}}(u,v) = \sum_{j=1}^{H}|u_j - v_j|.$$

6. **k-NN Classification:** Select the k closest training examples (distance tie broken by reading order, as in EP07_01) and classify by the majority class among them. In the case of a **voting tie** between two or more classes, choose the one that appears **first** in the class list from item 1.
7. **Confusion matrix:** Build a $C \times C$ matrix in which the row corresponds to the actual class and the column to the predicted class, following the class order from item 1.
8. **Accuracy:** Calculate the global accuracy as the ratio of correct predictions to Q.
9. **Output:** For each test sample, in input order, print the predicted class. Then print the confusion matrix (one row per actual class, values separated by spaces, in class order). Finally, print the accuracy rounded to 4 decimal places.

7.18.6.2 📌 Computational Constraints

- **Selectable metric:** implement both distances; the metric M defines which one is used throughout the execution (it is not possible to mix metrics in the same run).
- **Deterministic voting tie-break:** the criterion from item 6 (class list order) must be followed even when the tie involves more than two classes.
- **Training and test independence:** there is no need to validate that the test samples do not appear in training — assume the input is valid.

7.18.6.3 🧠 Theoretical Foundation

Exercise stage	Corresponding stage in the chapter
Training/test histograms already extracted	`descritor_lbp` applied to synthetic textures
Euclidean or Manhattan distance	`metric` parameter of the `KNeighborsClassifier`
Majority voting with k neighbors	`KNeighborsClassifier.predict`
$C \times C$ confusion matrix	`confusion_matrix` from `scikit-learn`
Global accuracy	`accuracy_score` from `scikit-learn`

This exercise highlights, in a controlled way, a result discussed in the chapter: the **choice of distance metric** and the **value of** k can change the predicted class for the same sample, even keeping the descriptor used fixed — reinforcing that, in classical pattern recognition, the descriptor, the metric, and the classifier form an interdependent system, not isolated parts.

7.18.6.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: integer C followed by C class names.
- Line 2: integer H, *string* M, and integer k.
- Line 3: integer N.
- Next N training lines: class name followed by H real values.
- Next line: integer Q.
- Next Q test lines: actual class name followed by H real values.

Output:

- Q lines with the predicted class of each test sample, in input order.
- C lines with the confusion matrix (one row per actual class).
- Last line: `Acuracia: <value>`.

7.18.6.5 📌 Examples

Input (summarized)	Output	Observation
2 granular listrada 2 euclidiana 1 4 granular 0.9 0.1 granular 0.8 0.2 listrada 0.1 0.9 listrada 0.2 0.8 2 granular 0.85 0.15 listrada 0.15 0.85	granular listrada 1 0 0 1 Acuracia: 1.0000	With $k = 1$, each test sample is classified by the closest training neighbor.

> **ℹ Note**
>
> This simulator uses a simplified set of **3 classes** (`granular`, `striped`, `spotted`) over fictional 2D points, solely to illustrate the voting, tie-breaking, and confusion matrix pipeline of k-NN. In **EP07_07**, you will apply this same logic to a real image mosaic, which introduces a fourth class (`checkered`) and replaces the 2D points with LBP histograms extracted directly from the image pixels.

```
%%writefile EP07_06.py
# Python code
```

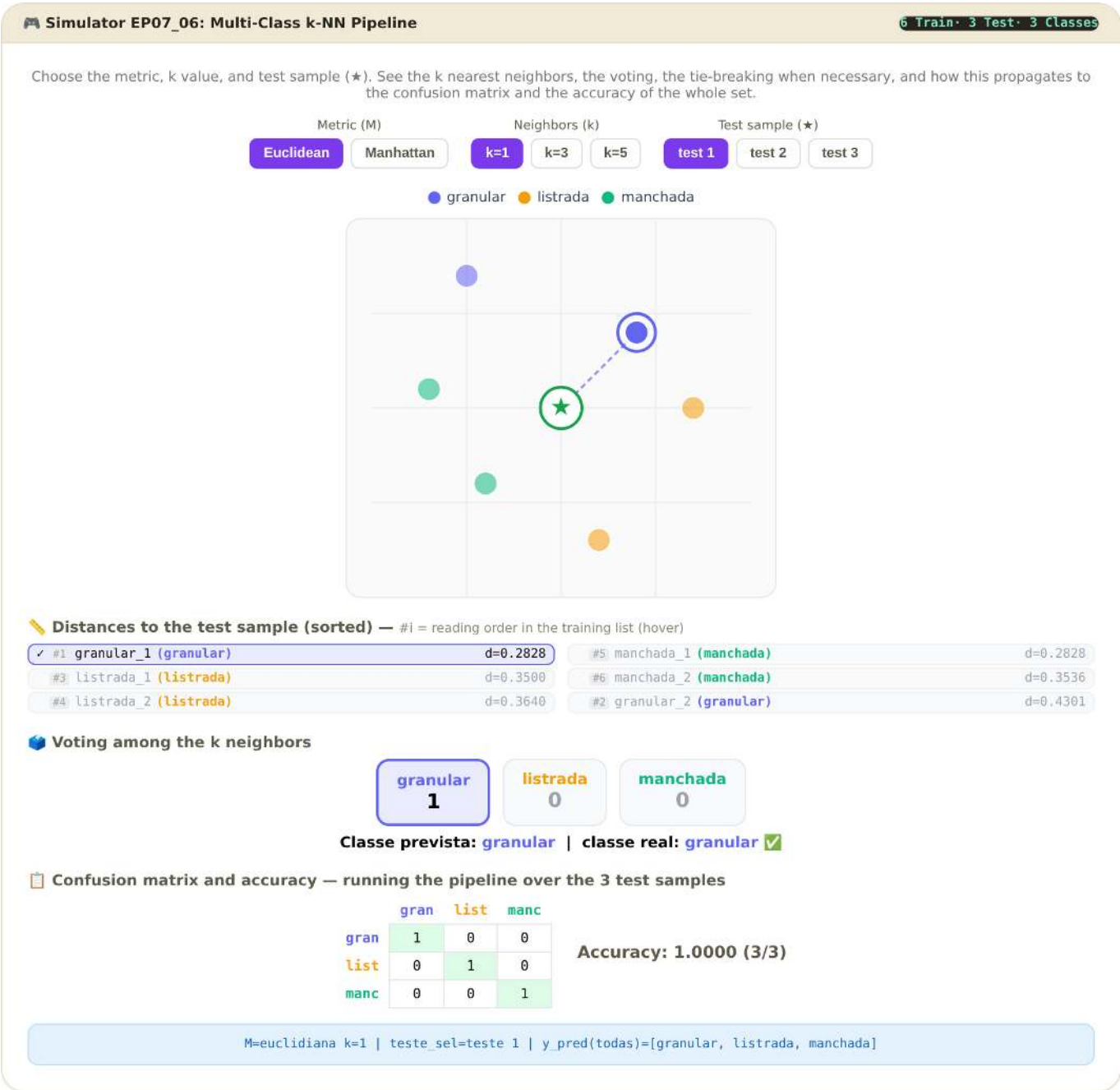
```
Overwriting EP07_06.py
```

```
TestSuite("EP07_06.py").run()
```

```
<IPython.core.display.HTML object>
```

7.18.7 EP07_07 ⚫ Real Classification of a Texture Mosaic via LBP + k-NN

In the previous exercises, the LBP descriptor (**EP07_04**) and the multiclass k-NN classifier (**EP07_06**) were studied separately, always based on data already provided as input — isolated 3×3 neighborhoods or previously extracted histograms. In this final exercise of the chapter, the program should **read a real image**, in **ASCII PGM (P2)** format, compute the LBP descriptor directly from the pixels, and then classify each region using k-NN, reproducing, on a reduced scale, the complete flow of a texture recognition system.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap07/sim-ep0706.html

Figure 7.26: Simulator EP07_06: k-NN Multi-Class *Pipeline* (voting, tie-breaking and confusion matrix)

This approach also anticipates the idea of **region mosaic classification**, related to semantic segmentation studied in a later chapter.

The interactive simulator from **EP07_06** used only three classes (`granular`, `striped`, and `blotchy`) represented by fictitious two-dimensional points. In this exercise, a fourth class, **checkered**, is added, and the points are replaced by LBP histograms extracted from a real image.

The input image is a **mosaic** formed by a $G \times G$ grid of square blocks of $S \times S$ pixels. Each block contains a sample of one of the four synthetic texture classes from the chapter: **granular**, **striped**, **blotchy**, or **checkered** (checkerboard pattern with alternating intensities). As in the other exercises in the book, image loading is performed by the didactic function `mm.readImg`.

> Why a single mosaic, rather than several images?
>
> The input gathers the $G \times G$ texture samples into a single **PGM** file, merely to simplify data reading and avoid opening multiple files. For the algorithm, this does not alter the processing: each block is treated independently, as if it were an isolated image. The only exception is the **border exclusion** (item 4 below).

7.18.7.1 📋 Implementation Guidelines

1. **Reading image dimensions**

   Read, from standard input, two lines containing, respectively, the number of rows L and the number of columns C of the mosaic (both multiples of the block size S, with $L = C$).

2. **Image loading**

   Use the didactic function

   ```
   f = mm.readImg(L, C)
   ```

   to read the $L \times C$ intensity values (grayscale, `uint8`) of the mosaic.

3. **Grid parameters**

   Read the integer G (number of blocks per side) and the integer S (side length of each block, in pixels), satisfying $L = C = G \times S$.

4. **LBP code computation per pixel**

   For each **interior** pixel of the image (i.e., one not on the global border of `f` — row or column 0 or $L - 1/C - 1$), compute the LBP code with $P = 8$ neighbors and radius $R = 1$, traversing the neighbors **clockwise** starting from the upper-left corner, exactly as in EP07_04: `[lin-1][col-1]`, `[lin-1][col]`, `[lin-1][col+1]`, `[lin][col+1]`, `[lin+1][col+1]`, `[lin+1][col]`, `[lin+1][col-1]`, `[lin][col-1]`.

   Pixels on the global border of the image **do not** have a complete neighborhood and should be **ignored** (they do not contribute to any histogram). This includes border pixels that fall within the interior of a block (exclusion is always relative to the border of the entire image, not to the border of each block).

5. **Uniform LBP histogram per block (10 bins)**

   For each block (i, j) of the grid $(i, j = 0, \dots, G - 1)$, accumulate, among its valid pixels (item 4), a histogram $H^{(i,j)}$ of 10 bins:

   - Considering the circular bit sequence $s_0, \dots, s_7$ of the pixel (same transition rule as in EP07_04): if the number of transitions is ≤ 2 (**uniform** pattern), the pixel contributes to bin $\text{popcount}(s_0, \dots, s_7) \in \{0, \dots, 8\}$ (number of bits equal to `1`);
   - Otherwise (**non-uniform** pattern), the pixel contributes to bin 9.

At the end, normalize each block's histogram by dividing by the number of valid pixels it contains, obtaining $\hat{H}^{(i,j)}$, with $\sum_{b=0}^{9} \hat{H}^{(i,j)}[b] = 1$.

6. **Training prototypes**

   Read the integer Ncl (number of classes) followed by Ncl class names (the order defining the confusion matrix and voting tie-break, as in EP07_06); then read the *string* M (metric: `euclidean` or `manhattan`) and the odd integer k; finally, read the integer N (number of prototypes) and, for each one, the class name followed by 10 real values (already normalized prototype histogram).

7. **k-NN classification of each block**

   For each block, compute the distance from $\hat{H}^{(i,j)}$ to each of the N prototypes, using metric M (same formulas as in EP07_06). Select the k closest prototypes (distance tie-break by prototype reading order) and classify by majority class (voting tie-break by the class order from item 6).

8. **True labels and evaluation**

   Read, on a single line, the $G \times G$ **true** class names of each block, in row-major order of the grid (block $(0,0)$, $(0,1)$, ..., $(0,G-1)$, $(1,0)$, ...). Build the $Ncl \times Ncl$ confusion matrix (row = true class, column = predicted class) and the overall accuracy.

9. **Output**

   Print, for each block (in the same reading order of the true labels from item 8), the predicted class. Then print the confusion matrix (one row per true class, in the order of item 6). Finally, print the accuracy, rounded to 4 decimal places.

7.18.7.2 📌 Computational Constraints

- **Fixed descriptor:** $P = 8$, $R = 1$, and 10 bins (per item 5) are fixed in this exercise — they are not read from the input.
- **Global border exclusion, not per-block:** a pixel at the boundary between two blocks, but within the image interior, is valid and contributes normally to the histogram of the block to which it belongs.
- **Reading order as tie-break criterion:** both distance tie-breaking (item 7) and voting tie-breaking (item 7) follow exactly the same conventions as EP07_01 and EP07_06.
- **Prototypes as input, not learned:** unlike Practical Project 2, the training histograms are provided directly as input; the program must not generate synthetic textures.

7.18.7.3 🧠 Theoretical Foundation

Exercise step	Corresponding step in the chapter
Image reading via `mm.readImg`	Image acquisition in the pattern recognition *pipeline*
LBP code per pixel (EP07_04)	`local_binary_pattern(image, P=8, R=1, method="uniform")`
10-bin histogram per block	`descritor_lbp` function from Practical Project 2 (`bins=10`, `range=(0, P+2)`)
k-NN classification with selectable metric (EP07_06)	`KNeighborsClassifier` trained on `X_textura`
$Ncl \times Ncl$ confusion matrix and accuracy	`confusion_matrix` and `accuracy_score` on `yt_teste`

This exercise highlights, with real pixels rather than synthetic values, a limitation discussed in the final section of the chapter: texture classes visually distinct to a human observer — such as **granular** and **blotchy** — may produce similar LBP histograms when the considered neighborhood is small ($R = 1$), since both exhibit a high frequency of non-uniform patterns at the scale of a single pixel. The **checkered** class, on the other hand, due to its regular and repetitive edges, tends to be separated more easily. The confusion matrix produced is expected to reflect precisely this pattern of partial confusion.

7.18.7.4 📦 Input and Output Specification (VPL)

Input:

```
L
C
[L x C image matrix]
G S
Ncl class_name_1 ... class_name_Ncl
M k
N
class_name h0 h1 ... h9      (repeated N times)
label(0,0) label(0,1) ... label(G-1,G-1)
```

Output:

- $G \times G$ lines with the predicted class of each block, in grid reading order.
- Ncl lines with the confusion matrix (one line per true class, values separated by spaces).
- Last line: `Accuracy: <value>`.

7.18.7.5 📌 Example (manual verification)

To check the descriptor implementation before testing it on a complete mosaic, consider a 6×6 **homogeneous** image, with all pixels at intensity 100, treated as a single block ($G = 1$, $S = 6$). Since every interior pixel has its 8 neighbors with intensity equal to the center ($g_p \geq g_c$ in all cases), all bits s_p are `1`, the number of transitions is 0 (uniform), and the bin is popcount(11111111) = 8. The histogram of the single block is therefore `0 0 0 0 0 0 0 0 1 0`.

Input (summarized)	Output	Observation
6 6 [36 values equal to 100] 1 6 2 uniform other euclidean 1 2 uniform 0 0 0 0 0 0 0 0 1 0 other 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 uniform	uniform 1 0 0 0 Accuracy: 1.0000	Distance from the block to the `uniform` prototype is exactly 0; the `other` class does not appear in the true label, so its row in the confusion matrix is null.

7.18.7.6 📌 Reference Files (.pgm)

For local debugging, two test mosaics in ASCII P2 format are provided (attached to this assignment; when integrating them into the chapter repository, save them in `all/cap07/data/EP07/`):

- 📥 **Case 1 — Simple mosaic (`Caso1_Mosaico_Simples.pgm`)**: 2×2 grid of 24×24-pixel blocks, one sample of each of the four classes, with low noise — useful for validating image reading and classification logic in a controlled scenario.
- 📥 **Case 2 — Mixed mosaic (`Caso2_Mosaico_Misto.pgm`)**: 3×3 grid of 16×16-pixel blocks, with repeated classes and greater variability — a scenario in which the confusion between **granular** and **blotchy** discussed in the Theoretical Foundation tends to manifest.

Figure 7.27 displays both mosaics for visual inspection before implementation.

```python
import os
import urllib.request
```

```python
import numpy as np

def garantir_e_baixar_arquivo(nome_arquivo):
    diretorio_local = "dados/EP07"
    caminho_local = os.path.join(diretorio_local, nome_arquivo)

    # Create the local directory if it does not exist
    if not os.path.exists(diretorio_local):
        os.makedirs(diretorio_local)

    # If the file does not exist locally, download from the remote repository
    if not os.path.exists(caminho_local):
        url_base = "https://raw.githubusercontent.com/fzampirolli/"
        url_base += "pdi-vc/master/all/cap07/dados/EP07"
        url_arquivo = f"{url_base}/{nome_arquivo}"
        print(f"Downloading {nome_arquivo} from GitHub...")
        try:
            urllib.request.urlretrieve(url_arquivo, caminho_local)
        except Exception as e:
            raise IOError(f"Erro ao baixar {nome_arquivo} do GitHub. ",
                          "Verifique a conexão ou a URL. Detalhes: {e}")

    return caminho_local

def ler_pgm_p2(caminho):
    with open(caminho) as f:
        linhas = [l for l in f.read().split() if l]
    assert linhas[0] == "P2"
    C, L = int(linhas[1]), int(linhas[2])
    maxv = int(linhas[3])
    valores = list(map(int, linhas[4:4 + L * C]))
    return np.array(valores, dtype=np.uint8).reshape(L, C)

# Ensures the download and obtains the correct path
arq_caso1 = garantir_e_baixar_arquivo("Caso1_Mosaico_Simples.pgm")
arq_caso2 = garantir_e_baixar_arquivo("Caso2_Mosaico_Misto.pgm")

# Reads the PGM matrices
caso1 = ler_pgm_p2(arq_caso1)
caso2 = ler_pgm_p2(arq_caso2)

mm.show(
    [caso1, caso2],
    titles=[
        "Case 1: Simple Mosaic\n(2x2 blocks, 1 sample/class)",
        "Case 2: Mixed Mosaic\n(3x3 blocks, repeated classes)",
    ],
    cols=2,
    figsize=(8, 4),
)
```

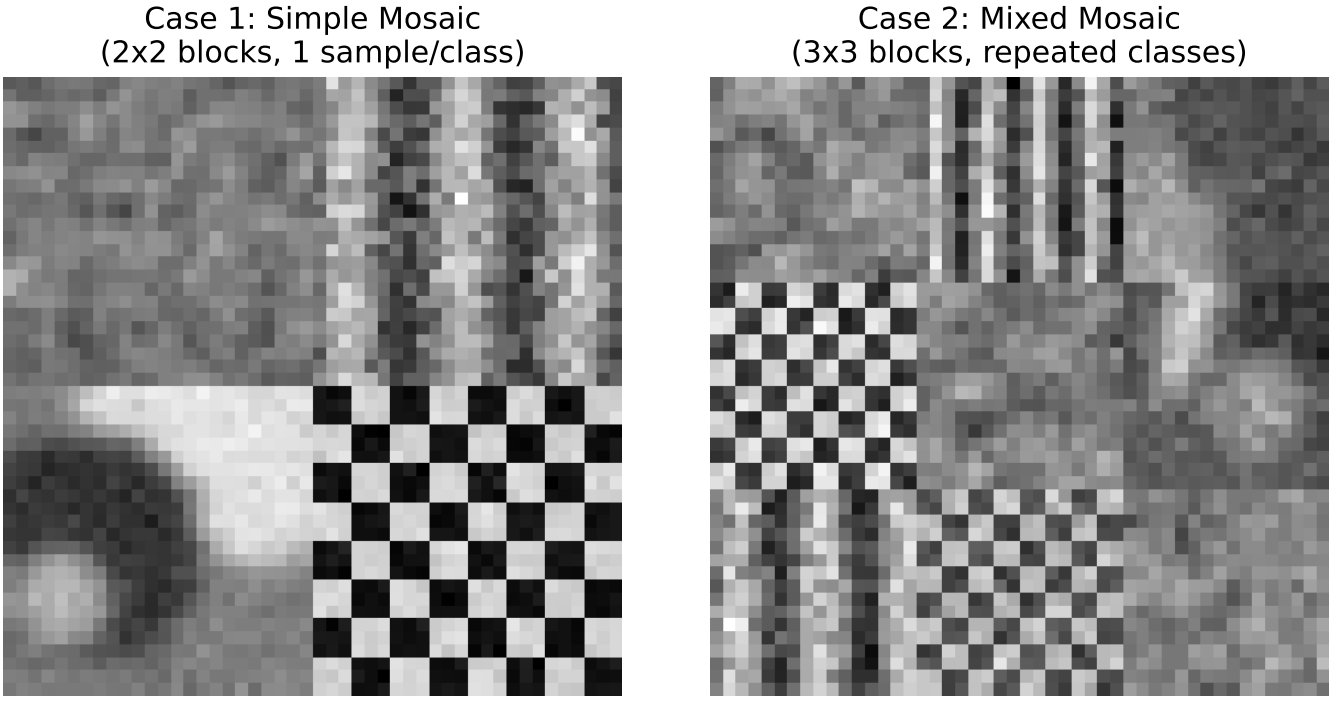
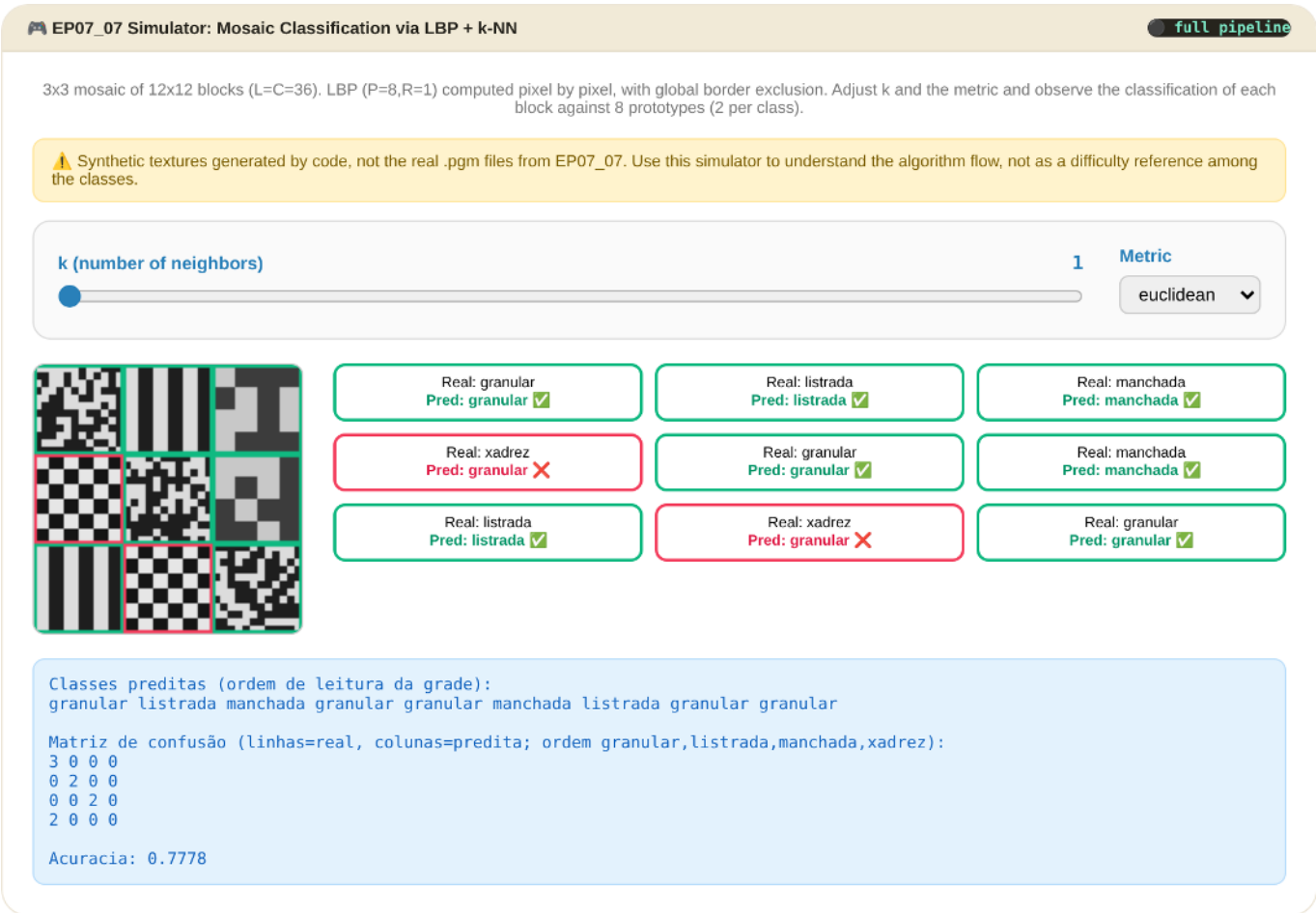


Figure 7.27: Reference mosaics (ASCII PGM format) used in EP07_07. Case 1: 2x2 grid with one sample of each class. Case 2: 3x3 grid with repeated classes and greater variability.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap07/sim-ep0707.html

Figure 7.28: Simulator EP07_07: Classification of a Texture Mosaic via LBP + k-NN

```
%%writefile EP07_07.py
# Python code
```

```
Overwriting EP07_07.py
```

```
TestSuite("EP07_07.py").run()
```

```
<IPython.core.display.HTML object>
```

Chapter 8

Understanding Scenes: Feature Matching, Detection, and Segmentation

capa08/images/thumbnail.jpg capa08/images/thumbnail.jpg

In **Chapter 7**, we studied how to represent images using descriptors and use these representations for classification tasks. In this chapter, the problem is broadened: in addition to recognizing patterns, it becomes necessary to establish correspondences between different images, automatically locate objects of interest, and interpret the spatial organization of a scene.

These problems constitute some of the main tasks in Computer Vision and represent a natural step after image classification. To solve them, classical methods for feature matching, object detection, and image segmentation will be presented, along with an overview of modern approaches based on *Deep Learning*, preparing the transition to **Chapter 9**.

8.1 Chapter Objectives

Upon completing this chapter, the student should be able to:

- **Establish correspondences between images** using local detectors and descriptors, estimating geometric transformations through homographies for registration and perspective correction;
- **Locate objects of interest** in images using classical detection methods and evaluate the results through metrics such as ***Intersection over Union*** **(IoU)** and the ***Non-Maximum Suppression*** **(NMS)** technique;
- **Differentiate segmentation paradigms** — semantic, instance, and panoptic — understanding that panoptic segmentation unifies semantic segmentation and instance segmentation, providing a more complete description of the scene, and apply classical segmentation methods;
- **Extract geometric and topological descriptors** from segmented objects and export them as structured annotations (CSV or YOLO format), validating the quality of the annotations through the IoU metric;
- **Relate classical correspondence, detection, and segmentation methods** with modern approaches based on *Deep Learning*, studied in **Chapter 9**.

Figure 8.1 presents an overview of the main concepts and the relationships among the topics covered in this chapter, serving as a conceptual map to guide the reading.

8.2 Environment Setup

```
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")
```

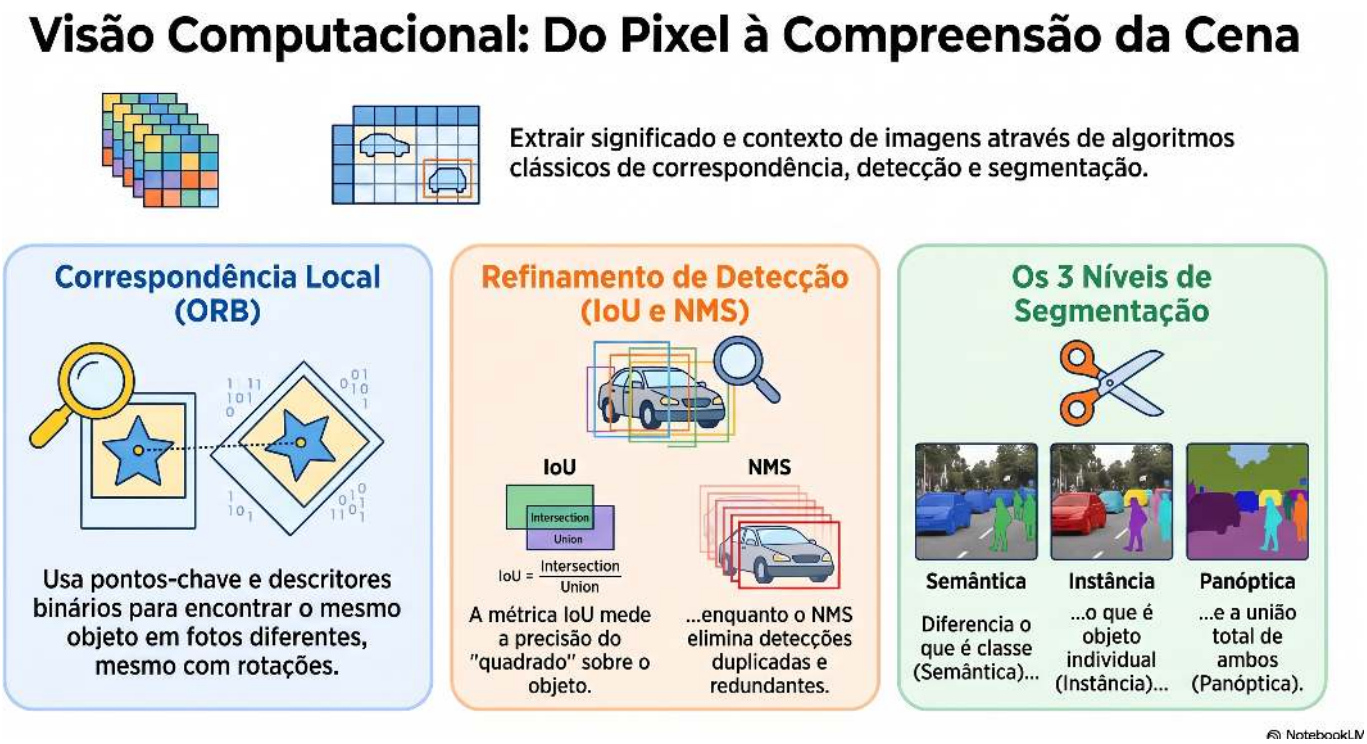


Figure 8.1: Overview of the main concepts covered in this chapter, including feature matching, object detection, image segmentation, and their relationships with modern approaches based on *Deep Learning*. **Source:** prepared with the assistance of Gemini Notebook ({GOOGLE}, 2025).

```
import config
config.setup(testsuite=True)
from morph import mm
from testsuite import TestSuite


import importlib
import subprocess
import sys

def setup_cap08():
    """Installs missing libraries required for this chapter."""
    pacotes = {
        "cv2": "opencv-python",
        "skimage": "scikit-image",
        "numpy": "numpy",
        "sklearn": "scikit-learn",
        "matplotlib": "matplotlib",
    }
    for mod, pkg in pacotes.items():
        if importlib.util.find_spec(mod) is None:
            resultado = subprocess.run(
                [sys.executable, "-m", "pip", "install", "-q", pkg]
            )
            if resultado.returncode != 0:
                print(f"[WARNING] Failed to install {pkg} (required for module {mod}).")


setup_cap08()

import cv2
import numpy as np
```

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from skimage import data as skdata
from skimage.filters import threshold_otsu
from skimage.measure import label
from skimage.morphology import remove_small_objects, opening, disk
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

8.3 Local Feature Detectors and Descriptors

In the previous chapter, each image was represented by a single feature vector used for classification. In several applications, however, it is necessary to compare only parts of the image, establishing correspondences between regions observed at different times, positions, or viewpoints. This task requires a local representation of the image, capable of identifying structures that are sufficiently distinct to be found again in other images.

This process is carried out in two complementary stages. Initially, a **detector** identifies points of interest (*keypoints*), usually associated with corners or regions with significant intensity variations. Subsequently, a **descriptor** numerically represents the neighborhood of each detected point, allowing corresponding regions to be compared between different images.

Once the pairs *(point, descriptor)* are obtained, matching consists of finding, for each descriptor in one image, the most similar descriptor in the other. These correspondences form the basis of several applications, such as image registration, three-dimensional reconstruction, visual navigation, and augmented reality.

8.3.1 The ORB Algorithm (*Oriented FAST and Rotated BRIEF*)

In this chapter, the **ORB** will be used, a local detector and descriptor that combines computational efficiency and robustness to rotations. The algorithm brings together three main components:

- **FAST** (*Features from Accelerated Segment Test*), responsible for keypoint detection;
- **BRIEF** (*Binary Robust Independent Elementary Features*), responsible for building the binary descriptor;
- a **neighborhood orientation estimation** mechanism, which makes the descriptor approximately invariant to rotation.

The **FAST** detector scans all pixels in the image. For each candidate pixel, it analyzes a circle of **16 pixels** around it. If a set of consecutive pixels exhibits intensity significantly higher or lower than the central pixel's intensity, that pixel is considered a **keypoint**. Subsequently, candidates that are too close are filtered out, preserving only the most representative ones.

After keypoint detection, the **BRIEF** descriptor is computed over a neighborhood around each keypoint, rather than only over the 16 pixels used by FAST. In this region, a **sampling pattern**, consisting of a fixed set of point pairs (x, y) distributed within a window around the keypoint, is used to perform intensity comparisons according to Equation 8.1. In ORB, the **predominant neighborhood orientation** is estimated from the intensity distribution in this region, and the sampling pattern is rotated according to this orientation. Furthermore, ORB employs an optimized version of BRIEF, called **rBRIEF** (*Rotated BRIEF*), in which point pairs are selected to produce more discriminative descriptors with low correlation among their bits.

$$\tau(p; x, y) = \begin{cases} 1, & \text{if } I(x) < I(y), \\ 0, & \text{otherwise}, \end{cases} \tag{8.1}$$

where:

- p is the keypoint detected by FAST;
- x and y are two points in the neighborhood of p, chosen by the BRIEF sampling pattern;

- $I(\cdot)$ represents the intensity of a pixel;
- $\tau(p; x, y)$ is the result of the binary comparison between points x and y.

Each comparison generates one bit of the descriptor. The concatenation of all these comparisons forms the binary descriptor associated with the keypoint.

Since the descriptor is binary, the similarity between two points is measured by the **Hamming distance**, corresponding to the number of differing bits between two descriptors. This metric can be computed very efficiently through logical operations on the bits, making ORB suitable for real-time applications.

8.3.2 Exploring the ORB Simulator

Figure 8.3 interactively illustrates how the **ORB** descriptor is constructed. Each segment represents one of the **32 point pairs** (x, y) used in Equation 8.1. The segment color indicates the result of the intensity comparison: **green** when $I(x) < I(y)$ (bit equal to 1) and **red** otherwise (bit equal to 0). The **yellow** arrow represents the predominant orientation of the neighborhood, estimated from the intensity centroid. The BRIEF sampling pattern is rotated according to this orientation, making the descriptor approximately invariant to rotation.

Initially, set the noise to zero and compare the bit sequence with the *slider* at 0° and then at 15° (record the value of the "Binary Descriptor Signature" field in each case):

```
0°  : 00000000000000100000011111100010
15° : 00000000000000100000011111100010
```

In this example, both descriptors are identical, and the "Hamming Dist." panel confirms a zero distance — evidence that the orientation estimation is correctly compensating for the image rotation.

Now click "Add Noise" and repeat the experiment. Since the noise is generated randomly on each run, the values below are only an example — yours will differ, but they should present a Hamming distance of similar magnitude (typically between 6 and 14 bits, out of a total of 32):

```
0°  : 01000000001000110110011111101010
15° : 10110000001011100100111111100010
```

The noise alters part of the intensity comparisons, modifying some bits of the descriptor. The difference between two descriptors is measured by the **Hamming distance**, corresponding to the number of positions where the bits differ. For the descriptors above, this distance equals **10**.

An efficient way to compute this distance in Python consists of applying the **XOR** operation (^), which identifies the differing bits, followed by the `bit_count()` method, which counts how many bits equal to 1 exist in the result.

```python
def hamming(a: int, b: int) -> int:
    return (a ^ b).bit_count()
a = 0b01000000001000110110011111101010
b = 0b10110000001011100100111111100010
hamming(a, b)
```

```
10
```

Figure 8.2 compares this implementation with a version based on character comparison and with the optimized OpenCV implementation (`cv2.NORM_HAMMING`).

```python
import random, timeit, cv2, numpy as np, pandas as pd
import matplotlib.pyplot as plt

BITS, N = 256, 100_000
A = [''.join(random.choice('01') for _ in range(BITS)) for _ in range(N)]
```

```
B = [''.join(random.choice('01') for _ in range(BITS)) for _ in range(N)]
Ai, Bi = map(lambda L: [int(x,2) for x in L], (A,B))
Acv = np.array([[int(s[i:i+8],2) for i in range(0,BITS,8)] for s in A], np.uint8)
Bcv = np.array([[int(s[i:i+8],2) for i in range(0,BITS,8)] for s in B], np.uint8)

H = [
    ("Strings", lambda: sum(sum(x!=y for x,y in zip(a,b)) for a,b in zip(A,B))),
    ("XOR+bit_count()", lambda: sum((a^b).bit_count() for a,b in zip(Ai,Bi))),
    ("OpenCV", lambda: sum(cv2.norm(a,b,cv2.NORM_HAMMING) for a,b in zip(Acv,Bcv)))
]

df = pd.DataFrame(
    [(n, timeit.timeit(f, number=1)) for n,f in H],
    columns=["Método","Tempo (s)"]
)
df["Speedup"] = (df.iloc[0,1]/df["Tempo (s)"]).round(1)

print(df)

plt.figure(figsize=(6,3))
plt.bar(df["Método"], df["Tempo (s)"])
plt.ylabel("Tempo (s)")
plt.show()
```

```
            Método  Tempo (s)  Speedup
0          Strings   1.474437      1.0
1  XOR+bit_count()   0.016943     87.0
2           OpenCV   0.232796      6.3
```

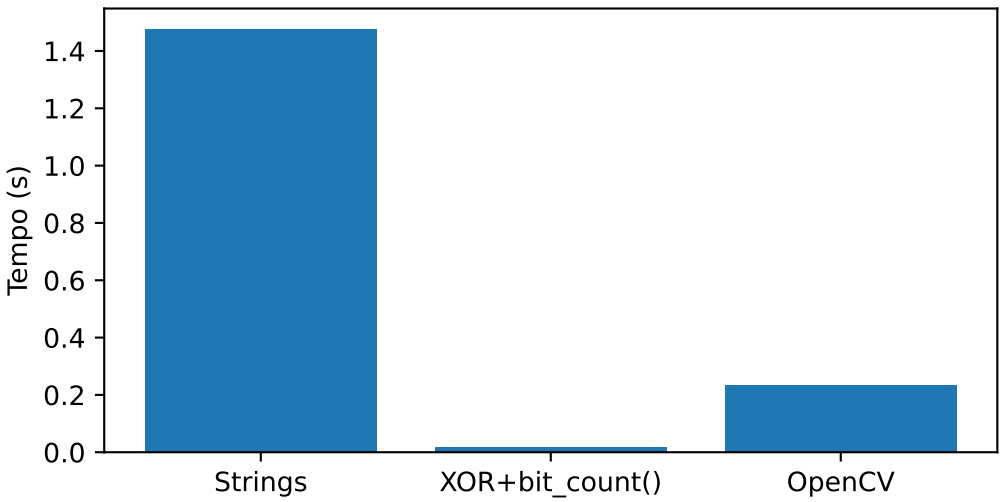


Figure 8.2: Comparison of performance of different implementations of the Hamming distance.

8.4 Feature Matching and Homography with ORB

To illustrate the feature matching process, a synthetic scene is used, obtained by rotating the original image by 20°. This transformation simulates a second capture of the same scene from a different viewpoint. Figure 8.4 presents the reference image and its rotated version, which will be used in the following steps.

```
caminho_local = "dados/fig-08-Gemini-faces3.png"
url_imagem = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/all/cap08/"
url_imagem += caminho_local
```

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-08-orb.html

Figure 8.3: Interactive simulator of the ORB descriptor: explore the rotation logic and binary descriptor construction. Change the rotation to observe how the BRIEF binary test sampling pattern (green and red lines) dynamically orients itself to ensure angular invariance.

```
if not os.path.exists(caminho_local):
    mm.write(mm.read(url_imagem), caminho_local)

img_original = mm.read(caminho_local)
img_cena = mm.rotate(img_original, angle=20)

mm.show([img_original, img_cena],
        titles=["Original", "Rotation (20°)"],
        cols=2, figsize=(8, 4))
```

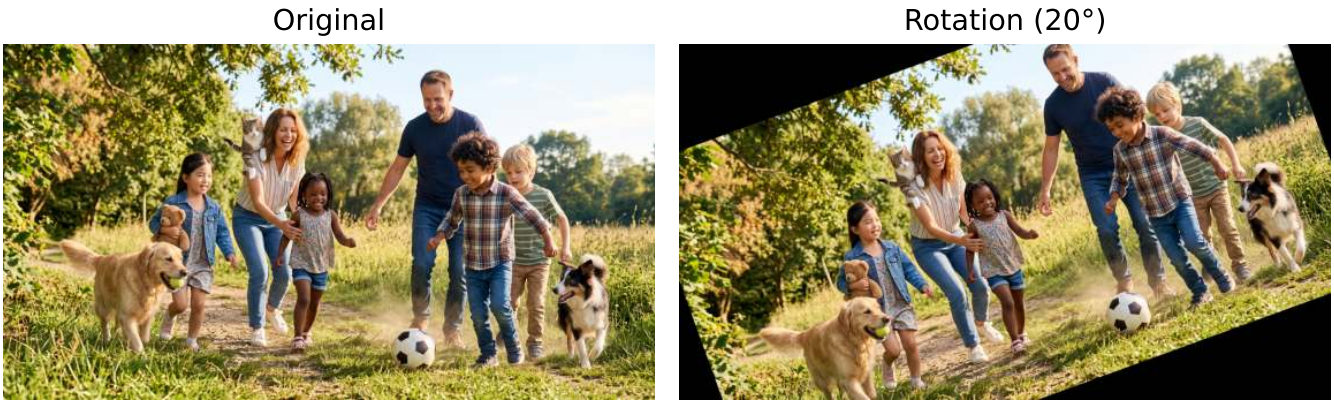


Figure 8.4: Original image generated by Gemini and rotated version (20°), simulating a change of viewpoint.

8.4.1 Detecting and Matching Features with ORB

With both images available, **ORB** detects keypoints and computes their binary descriptors. Next, the **BFMatcher** from the `cv2` library establishes correspondences between descriptors using the Hamming distance and cross-check validation. Finally, the matches are sorted from the smallest to the largest Hamming distance, prioritizing potentially more reliable pairs. In the following code, which generates Figure 8.5, the following steps stand out:

1. `cv2.ORB_create(nfeatures=500)` instantiates the ORB detector, limiting the search to the **500 most representative keypoints** of each image. This restriction reduces computational cost and avoids selecting points with little distinctiveness.
2. `orb.detectAndCompute(...)` performs, in a single call, keypoint detection via **FAST** and descriptor computation via **oriented BRIEF**, returning the list of keypoints (`kp`) and their **256-bit** binary descriptors (`des`).
3. `cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)` creates a brute-force matcher that uses the **Hamming distance** — the same metric explored in Figure 8.3 — to compare each descriptor from the original image with all descriptors from the rotated image. The `crossCheck=True` parameter retains only pairs in which the best match is reciprocal, that is, when the best match for A is B and, simultaneously, the best match for B is A. This criterion eliminates a large portion of ambiguous matches.
4. `matches = sorted(...)` sorts the matches from the smallest to the largest Hamming distance. The smaller this distance, the greater the similarity between descriptors and, consequently, the higher the probability that the match is correct.

Figure 8.5 displays only the five matches with the smallest Hamming distance. Although these pairs are the most promising, there is still no geometric constraint between corresponding points. As a result, some links may represent false matches, justifying the use of **RANSAC** in the next step to identify only geometrically consistent correspondences.

```python
orb = cv2.ORB_create(nfeatures=500)
kp1, des1 = orb.detectAndCompute(img_original, None)
kp2, des2 = orb.detectAndCompute(img_cena, None)
print(f"Keypoints detected: {len(kp1)} (original), {len(kp2)} (scene)")

bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = sorted(
    bf.match(des1, des2),
    key=lambda m: m.distance
)
print(f"Matches found: {len(matches)}")

def draw_matches_destacado(img1, kp1, img2, kp2, matches, espessura=2, raio_ponto=4,
                           seed=42, cor_fixa=None):
    """Draws the images side by side with lines connecting the corresponding points.
    If cor_fixa=None, each match receives a random color (making it easier to distinguish
    individual links). If cor_fixa is defined (e.g., green), all lines use the
    same color - useful for highlighting a specific subset, such as the RANSAC inliers.
    """
    h1, w1 = img1.shape[:2]
    h2, w2 = img2.shape[:2]
    h = max(h1, h2)
    canvas = np.zeros((h, w1 + w2, 3), dtype=np.uint8)
    canvas[:h1, :w1] = cv2.cvtColor(img1, cv2.COLOR_GRAY2BGR) if img1.ndim == 2 else img1
    canvas[:h2, w1:w1+w2] = cv2.cvtColor(img2, cv2.COLOR_GRAY2BGR) if img2.ndim == 2 else img2

    rng = np.random.RandomState(seed)  # fixed seed = reproducible colors on each run

    for m in matches:
        pt1 = tuple(np.round(kp1[m.queryIdx].pt).astype(int))
        pt2 = tuple(np.round(kp2[m.trainIdx].pt).astype(int) + np.array([w1, 0]))

        cor = cor_fixa if cor_fixa is not None else \
            tuple(int(c) for c in rng.randint(60, 256, size=3))

        cv2.line(canvas, pt1, pt2, cor, espessura, lineType=cv2.LINE_AA)
        cv2.circle(canvas, pt1, raio_ponto, cor, -1, lineType=cv2.LINE_AA)
        cv2.circle(canvas, pt2, raio_ponto, cor, -1, lineType=cv2.LINE_AA)

    return canvas

# try the 5 worst: matches[-5:]
img_matches = draw_matches_destacado(
    img_original, kp1, img_cena, kp2, matches[:5],
    espessura=10, raio_ponto=15
)
mm.show([img_matches], titles=["Top 5 ORB Matches"], cols=1, figsize=(10, 5))
```

```
Keypoints detected: 500 (original), 500 (scene)
Matches found: 300
```

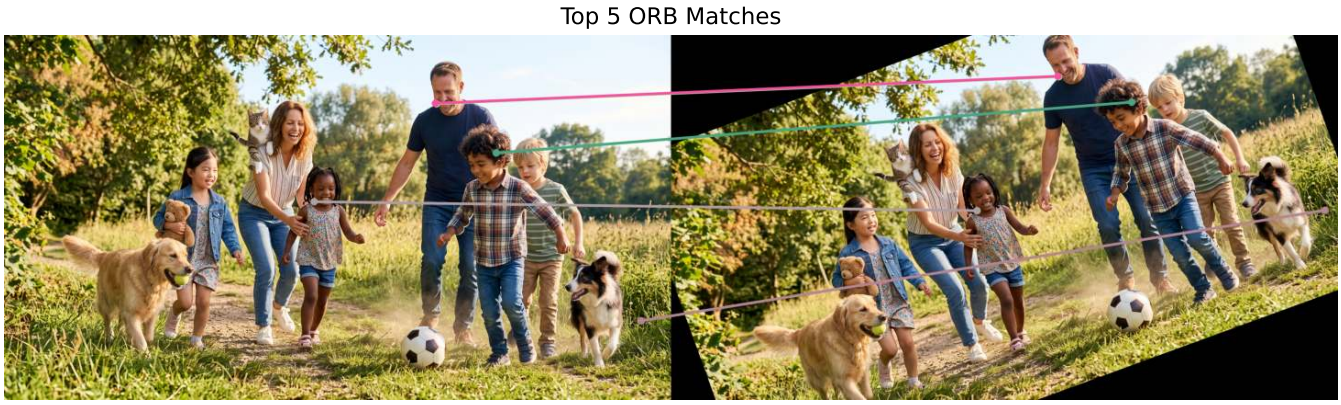


Figure 8.5: The 5 best ORB feature matches between the original image and the synthetic scene, before RANSAC filtering.

The function `draw_matches_destacado()` serves only a visualization purpose: it places the images side by side and draws lines between the corresponding pairs, without interfering with the estimation of correspondences.

8.5 Mathematical Modeling: Homography and RANSAC

8.5.1 Homography

In Exercises 10 and 11 of Chapter 2, the functions `cv2.getPerspectiveTransform` and `cv2.warpPerspective` were used to correct the perspective of images based on **four pairs of corresponding points** provided manually. In this chapter, these correspondences are now obtained automatically by ORB, making it possible to estimate the transformation between two images without user intervention.

Mathematically, this transformation is described by a **homography**, represented by a 3×3 matrix that relates the coordinates of the same plane observed from different viewpoints:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

After normalizing the homogeneous coordinates, the corresponding point is obtained:

$$\left(\frac{x'}{w'}, \frac{y'}{w'} \right).$$

Since the homography is defined up to a scale factor, it has **eight degrees of freedom**. Consequently, at least **four pairs of corresponding points** are required to estimate its parameters.

In practice, however, correspondences automatically produced by ORB may contain incorrect associations (*outliers*). To estimate the homography reliably even in the presence of such errors, the **RANSAC** algorithm is used, which is presented in the next section.

8.5.2 RANSAC

RANSAC (*Random Sample Consensus*) is a robust estimation algorithm capable of fitting a geometric model even in the presence of incorrect observations (*outliers*). In this chapter, the model of interest is a **homography**, estimated from the correspondences produced by ORB.

In each iteration, the algorithm:

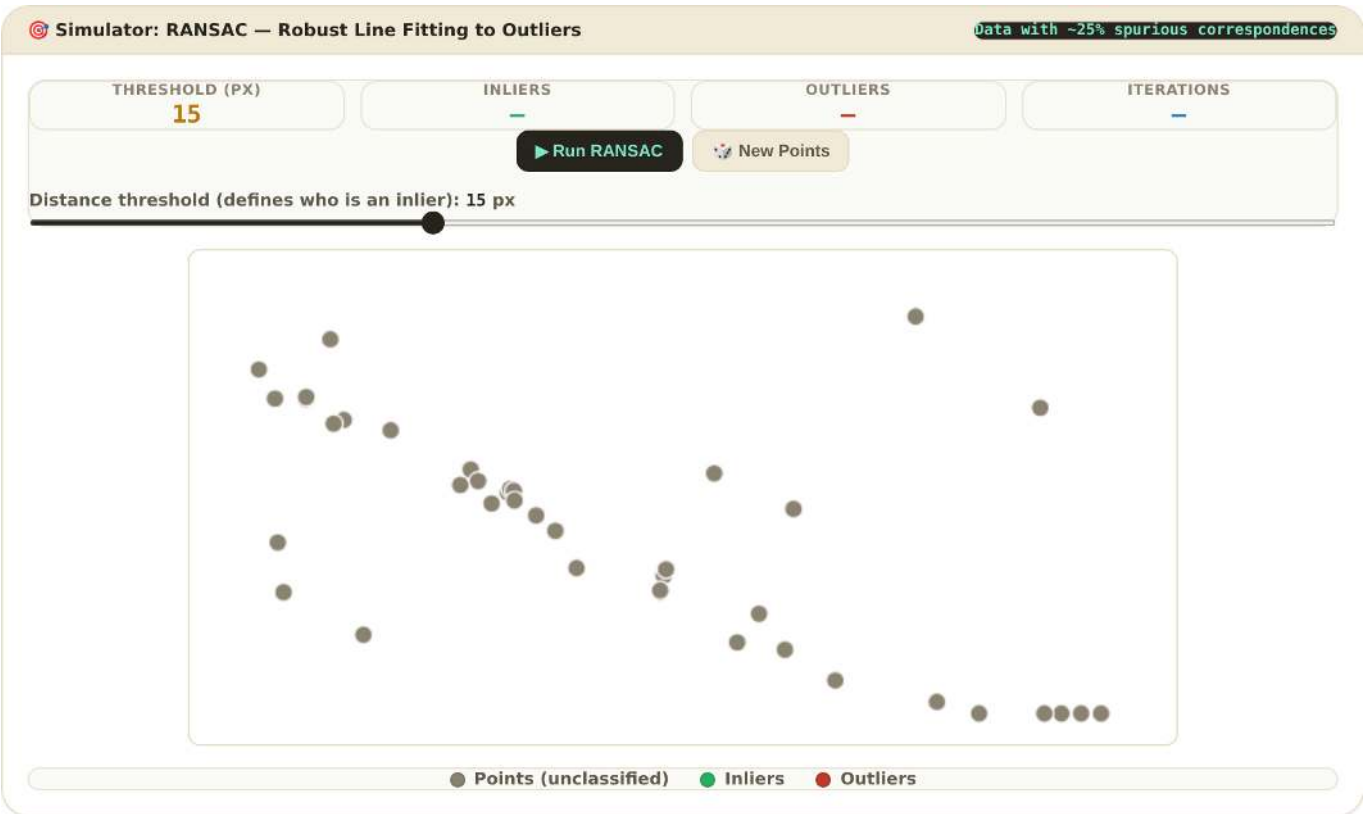
1. randomly selects a small subset of correspondences (four point pairs, in the case of the homography);

2. estimates a candidate homography from that subset;
3. checks which correspondences are compatible with this transformation, classifying them as ***inliers*** or ***outliers***;
4. records the homography that yields the largest number of ***inliers***;
5. re-estimates the homography using only the ***inliers*** found.

Although the example in this chapter uses a homography, RANSAC is a general-purpose algorithm and can be employed to estimate various geometric models, such as lines, circles, planes, and other transformations. In all cases, the principle is the same: generate candidate models from small random samples and select the one that exhibits the greatest consensus among the data.

To understand this process gradually, two simulators are presented.

The first, shown in Figure 8.6, uses the simplest possible example: fitting a **line** to a set of points containing approximately **25%** ***outliers***. The goal is to understand the fundamental steps of the algorithm — sampling, estimating a model, identifying ***inliers***, and repeating the process — without the complexity of image registration.



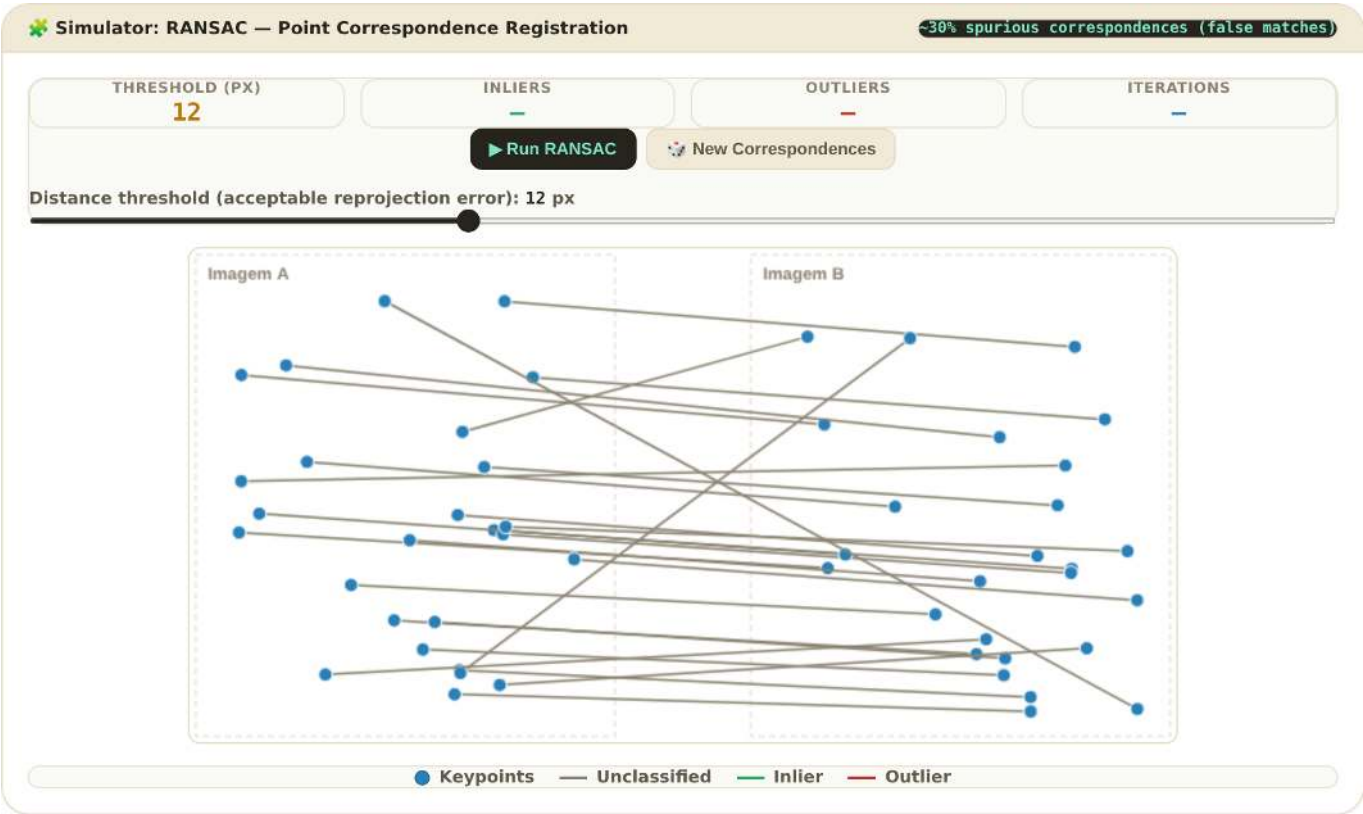
Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-08-ransac.html

Figure 8.6: Interactive simulator of the RANSAC algorithm: adjust the distance threshold and run the algorithm to observe the separation between *inliers* and *outliers*.

The second simulator, presented in Figure 8.7, approximates the problem studied in this chapter. Instead of a single set of points, **two images** containing correspondences between keypoints are considered. Some correspondences are correct (*inliers*), while others are incorrect (*outliers*), resulting from errors in the descriptor matching process. In this simulator, the estimated model is a **similarity transformation** (rotation, scale, and translation), simpler than a full homography but sufficient to illustrate the image registration problem.

In both simulators, the algorithm executed follows exactly the same principle subsequently used to estimate the homography. The only difference lies in the fitted geometric model.

Adjust the distance threshold and run the algorithm in each simulator to observe how RANSAC identifies the ***inliers***, discards the ***outliers***, and estimates a consistent model using only the valid correspondences.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-08-ransac-registro.html

Figure 8.7: Interactive simulator of RANSAC applied to image registration: key points from two images are matched by a descriptor, some correspondences are spurious (*outliers*), and RANSAC estimates the similarity transformation that aligns most of them.

8.5.3 Estimating Homography with RANSAC

After obtaining the correspondences between key points using ORB, the next step is to estimate the **homography** between the two images. For this purpose, the function `cv2.findHomography()` is used, which employs the **RANSAC** algorithm to compute this transformation and return a mask indicating which correspondences were classified as ***inliers***.

The following code performs four main operations:

1. extracts the coordinates of the corresponding points in each image;
2. estimates the homography using `cv2.findHomography(..., cv2.RANSAC)`;
3. receives the mask produced by RANSAC, in which each correspondence is classified as either an ***inlier*** or an ***outlier***;
4. uses this mask to select only the correspondences classified as ***inliers***.

Figure 8.8 presents only the correspondences classified as ***inliers***. It is observed that these pairs of points are compatible with the same geometric transformation, while inconsistent correspondences (*outliers*) are discarded. Consequently, the estimated homography represents the geometric relationship between the two images more faithfully.

```
pts1 = np.float32([kp1[m.queryIdx].pt for m in matches])
pts2 = np.float32([kp2[m.trainIdx].pt for m in matches])

H, mascara_inliers = cv2.findHomography(pts1, pts2, cv2.RANSAC, ransacReprojThreshold=5.0)
n_inliers = int(mascara_inliers.sum())
print(f"Estimated homography matrix:\n{H}\n")
print(f"Inliers: {n_inliers} of {len(matches)} correspondences \
      ({100*n_inliers/len(matches):.1f}%)")
```

```
matches_inliers = [m for m, ok in zip(matches, mascara_inliers.ravel()) if ok]

img_inliers = draw_matches_destacado(
    img_original, kp1, img_cena, kp2, matches_inliers,
    espessura=2, raio_ponto=4, cor_fixa=(0, 200, 0)  # green (BGR)
)
mm.show([img_inliers], titles=[f"Inlier Correspondences (RANSAC) - \
                               {n_inliers}/{len(matches)}"], cols=1, figsize=(10, 5))
```

```
Estimated homography matrix:
[[ 9.39407240e-01  3.41706397e-01 -1.77390658e+02]
 [-3.41929137e-01  9.39717641e-01  5.27534018e+02]
 [-1.12175998e-07 -4.86089683e-08  1.00000000e+00]]

Inliers: 285 of 300 correspondences       (95.0%)
```

Figure 8.8: Correspondences classified as *inliers* (green) by RANSAC when estimating the homography between the two images.

8.5.4 Registering the Image

After estimating the homography, the next step consists of using it to register the scene image in the coordinate system of the original image. This process allows aligning the two images, facilitating comparison between them.

The code performs three main operations:

1. applies the projective transformation using `cv2.warpPerspective()`, with the `cv2.WARP_INVERSE_MAP` option, which internally applies the inverse transformation without the need to explicitly compute H^{-1};
2. calculates the absolute pixel-by-pixel difference between the registered image and the original image using `cv2.absdiff()`;
3. displays the original image, the rotated scene, the registered image, and a map of the differences between the two images.

Figure 8.9 presents the result of the registration. It is observed that the registered image becomes visually very close to the original image, indicating that the estimated homography was able to correctly align the two views of the same scene. The difference map highlights only the regions where small discrepancies still exist, arising from interpolation errors, quantization, and the homography estimation itself.

```
h, w = img_original.shape[:2]

# WARP_INVERSE_MAP: applies H "from back to front", avoiding the manual calculation of H^-1
img_registrada = cv2.warpPerspective(img_cena, H, (w, h), flags=cv2.WARP_INVERSE_MAP)
```

```
erro = cv2.absdiff(img_original, img_registrada)

mm.show( # shows mm.gray(error)>10 in gray levels
    [img_original, img_cena, img_registrada, mm.gray(erro)>10],
    titles=["Original", "Scene (rotated)", "Registered", "Difference"],
    cols=2,
    figsize=(14, 8),
)
```

Figure 8.9: Recording the scene image using the inverse homography estimated by RANSAC.

Why does it work? — Robustness through consensus

Many estimation methods fit a model using **all** available observations, seeking to minimize the total error between the observed data and the fitted model (an approach known as **least squares**). When *outliers* are present, these incorrect observations can significantly shift the resulting outcome.
RANSAC follows a different strategy. Instead of using all data simultaneously, it successively estimates models from small random samples. Each model is then evaluated by the number of correspondences compatible with it. At the end of the iterations, the model that exhibits the greatest consensus among the data is selected, that is, the one with the largest number of ***inliers***.
Two parameters play a fundamental role in the algorithm:

- the **reprojection threshold**, which defines the maximum distance for a correspondence to be classified as an ***inlier***;
- the **number of iterations**, which must be sufficiently large to increase the probability of selecting at least one sample free of ***outliers***.

Although quite robust, RANSAC assumes that a predominant geometric model exists in the data. Its performance tends to decrease when the proportion of ***inliers*** is very small or when different geometric structures coexist in the same scene, making it difficult to identify a single dominant model.

8.6 Object Detection: *Haar Cascade* (Viola-Jones)

Feature matching answers the question: **"where is the same object or pattern previously observed?"**. **Object detection** solves a more general problem: automatically locating instances of a **category** (e.g., human faces), even if the specific objects have never been observed during training. While ORB requires two images to establish correspondences between points, **Haar Cascade** operates on a single image, directly identifying candidate regions that may contain the sought object.

The ***Haar Cascade*** algorithm, proposed by Viola and Jones [Viola; Jones (2001); Viola (2004)], combines Haar features, integral images, and a cascade of classifiers to perform object detection efficiently. Although more recent methods based on convolutional neural networks currently exist, Haar Cascade remains available in the OpenCV library and constitutes a classic example for the study of object detection techniques.

Its operation is based on three main components:

1. **Haar features:** simple rectangular filters that measure intensity differences between neighboring regions of the image, exploiting **characteristic contrast patterns** of the object, such as the eye region generally being darker than the forehead;
2. **Integral image:** data structure that allows fast computation of the sum of pixels in any rectangular region of the image:
$$I_{\text{integral}}(x, y) = \sum_{x' \leq x,\, y' \leq y} I(x', y'),$$
significantly reducing the computational cost of evaluating Haar features. With this structure, the sum of pixels in any rectangle can be obtained with only four accesses to the integral image;
3. **Cascade of classifiers:** during training, the AdaBoost algorithm selects and combines simple classifiers into a sequence of stages. During detection, regions that clearly **do not correspond to the object** are discarded in the early stages, while only the most promising candidates pass through the subsequent, more accurate and computationally more expensive stages. This strategy enables efficient search at different positions and scales of the image.

The following code, which generates Figure 8.10, uses previously trained classifiers provided by OpenCV to detect **faces** and then restricts the search for **eyes** only to the interior of each detected face. This strategy reduces false positives and decreases computational cost, as it avoids performing the eye search across the entire image.

During detection, a window traverses the image at different positions and scales. The `detectMultiScale()` function performs this search automatically. The `scaleFactor` parameter controls the reduction factor between consecutive window scales, while `minNeighbors` defines the minimum number of neighboring detections required to confirm an object, reducing spurious detections. The `minSize` parameter establishes the smallest object size considered during the search.

The function returns a list of rectangles, each described by the coordinates of the upper-left corner and the dimensions `(x, y, width, height)`. These rectangles delimit the regions classified as objects by the detector and are used to draw the boxes shown in Figure 8.10.

```
def get_cascade(nome):
    """Downloads (if necessary) and loads an OpenCV Haar Cascade classifier."""
    caminho = f"haarcascades/{nome}"
    os.makedirs("haarcascades", exist_ok=True)
    if not os.path.exists(caminho):
      url = f"https://raw.githubusercontent.com/opencv/opencv/master/data/haarcascades/{nome}"
      urllib.request.urlretrieve(url, caminho)
    return cv2.CascadeClassifier(caminho)

# Most used
#face_cascade = get_cascade("haarcascade_frontalface_default.xml")

# More accurate, but slower
```

```
face_cascade = get_cascade("haarcascade_frontalface_alt2.xml")

# Trade-off between speed and accuracy
#face_cascade = get_cascade("haarcascade_frontalface_alt.xml")

# Very fast, but less accurate
#face_cascade = get_cascade("haarcascade_frontalface_alt_tree.xml")

# for the eyes, most used
#eye_cascade = get_cascade("haarcascade_eye.xml")
eye_cascade  = get_cascade("haarcascade_eye_tree_eyeglasses.xml")

img_original = mm.read(caminho_local)

# Grayscale + histogram equalization (Chapter 3), as the Haar Cascade expects
img_rgb = mm.rotate(img_original, angle=0)
img_gray = cv2.equalizeHist(cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY))

# Detects faces and, within each, tries to detect the eyes
faces = face_cascade.detectMultiScale(
    img_gray,
    scaleFactor=1.1,      # Larger steps between scales: faster, but less sensitive
    minNeighbors=3,       # Minimum number of overlapping detections to confirm a face
    minSize=(90, 90)      # Ignores candidate regions smaller than 30×30 px
)

img_anotada = img_rgb.copy()
for (x, y, w, h) in faces:
    cv2.rectangle(img_anotada, (x, y), (x + w, y + h), (0, 255, 0), 3)
    olhos = eye_cascade.detectMultiScale(
        img_gray[y:y+h, x:x+w],
        scaleFactor=1.02,
        minNeighbors=4,
        minSize=(15, 15)
    )
    for (ex, ey, ew, eh) in olhos:
        cv2.rectangle(img_anotada, (x+ex, y+ey), (x+ex+ew, y+ey+eh), (255, 0, 0), 4)

print(f"Regions detected as face: {len(faces)}")
mm.show([img_anotada], titles=
        ["Haar Cascade detection: faces (green) and eyes (red)"],
        cols=1, figsize=(8, 8))
```

```
Regions detected as face: 5
```

Haar Cascade detection: faces (green) and eyes (red)

Figure 8.10: Face and eye detection with *Haar Cascade*.

Why does it work? — And why it also fails

In the image used in this example, the classifier detects the face and eyes, but it may also mark a second region over part of the image's background as if it were a face. This is an example of a **false positive**: the local distribution of intensities in that region is sufficiently similar to the patterns learned during training for the cascade to mistakenly classify it as a face.
This behavior highlights one of the main limitations of *Haar Cascade*. Since the method bases its decision solely on Haar features, i.e., intensity differences between rectangular regions, it does not explicitly represent the shape or meaning of objects present in the image. Thus, textures and contrast patterns similar to those found in faces can produce incorrect detections. Moreover, its performance tends to decline in the presence of large variations in pose, occlusions, facial expressions, and lighting conditions different from those found in the training data.
Despite these limitations, *Haar Cascade* remains useful in applications that prioritize low computational cost. In situations requiring greater generalization capability under variations in object appearance, modern methods based on deep neural networks tend to perform better.

8.7 Object Detection: Bounding Boxes, IoU, and NMS

Although object detection methods employ quite different strategies—from *Haar Cascade* to modern detectors based on deep neural networks—their results are typically represented by **bounding boxes**, defined by the coordinates $(x_{min}, y_{min}, x_{max}, y_{max})$.

8.7.1 *Intersection over Union* (IoU)

The ***Intersection over Union*** **(IoU)** metric quantifies the overlap between two bounding boxes, for example, the detection produced by an algorithm and the reference annotation (*ground truth*):

$$\text{IoU}(A, B) = \frac{\text{área}(A \cap B)}{\text{área}(A \cup B)} \in [0, 1].$$

The closer to 1, the greater the agreement between the boxes; the value 0 indicates no overlap. In evaluations of object detectors, it is common to consider a detection correct when IoU $\geq 0{,}5$, although specific applications may adopt different thresholds.

The IoU metric is not only used to evaluate detectors. It also constitutes the criterion employed by the **Non-Maximum Suppression** algorithm to decide when two bounding boxes represent the same object and, therefore, one of them must be eliminated.

8.7.2 Non-Maximum Suppression (NMS)

During detection, it is common for several bounding boxes to be associated with the same object. **Non-Maximum Suppression** (NMS) eliminates this redundancy in three steps:

1. sorts the detections by confidence score, from highest to lowest;
2. keeps the box with the highest confidence and eliminates those whose IoU with it exceeds a given threshold;
3. repeats the process with the remaining boxes until no relevant overlaps remain.

The example in Figure 8.11 illustrates this procedure with two objects, each initially represented by several overlapping boxes. In this example, each bounding box receives a **confidence score**, which represents the detector's degree of confidence that that region contains the sought object.

Instead of implementing the algorithm manually, one uses the `cv2.dnn.NMSBoxes` function, which implements the Non-Maximum Suppression employed in several modern detectors. The `desenhar_caixas` function displays, side by side, the boxes before and after NMS, evidencing the reduction from five detections to only two, preserving only the highest-confidence box for each object.

```
def desenhar_caixas(caixas, pontuacoes, indices, cor, tamanho=(450, 450)):
    """Draws the boxes (and their scores) indicated in `indices` on a white background."""
    tela = np.full((tamanho[1], tamanho[0], 3), 255, dtype=np.uint8)
    for i in indices:
        x0, y0, x1, y1 = caixas[i]
        cv2.rectangle(tela, (x0, y0), (x1, y1), cor, 2)
        cv2.putText(tela, f"{pontuacoes[i]:.2f}", (x0, y0 - 8),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, cor, 1, cv2.LINE_AA)
    return tela


caixas = np.array([
    [50, 50, 150, 150], [60, 55, 155, 145], [58, 60, 160, 150],
    [300, 300, 400, 420], [310, 305, 395, 415],
])
pontuacoes = np.array([0.90, 0.75, 0.60, 0.95, 0.70])

# cv2.dnn.NMSBoxes expects boxes in the format (x, y, width, height)
caixas_xywh = np.column_stack([
    caixas[:, 0], caixas[:, 1],
    caixas[:, 2] - caixas[:, 0], caixas[:, 3] - caixas[:, 1]
])

mantidas = cv2.dnn.NMSBoxes(
    bboxes=caixas_xywh.tolist(),
    scores=pontuacoes.tolist(),
    score_threshold=0.0,   # no cut by minimum confidence here
    nms_threshold=0.5      # IoU threshold to consider two boxes redundant
).flatten()
```

```
img_antes = desenhar_caixas(caixas, pontuacoes, range(len(caixas)), cor=(220, 38, 38))
img_depois = desenhar_caixas(caixas, pontuacoes, mantidas, cor=(22, 163, 74))

mm.show(
    [img_antes, img_depois],
    titles=[f"Before NMS ({len(caixas)} boxes)", \
            f"After NMS ({len(mantidas)} box(es))"],
    cols=2, figsize=(9, 4.5)
)
```

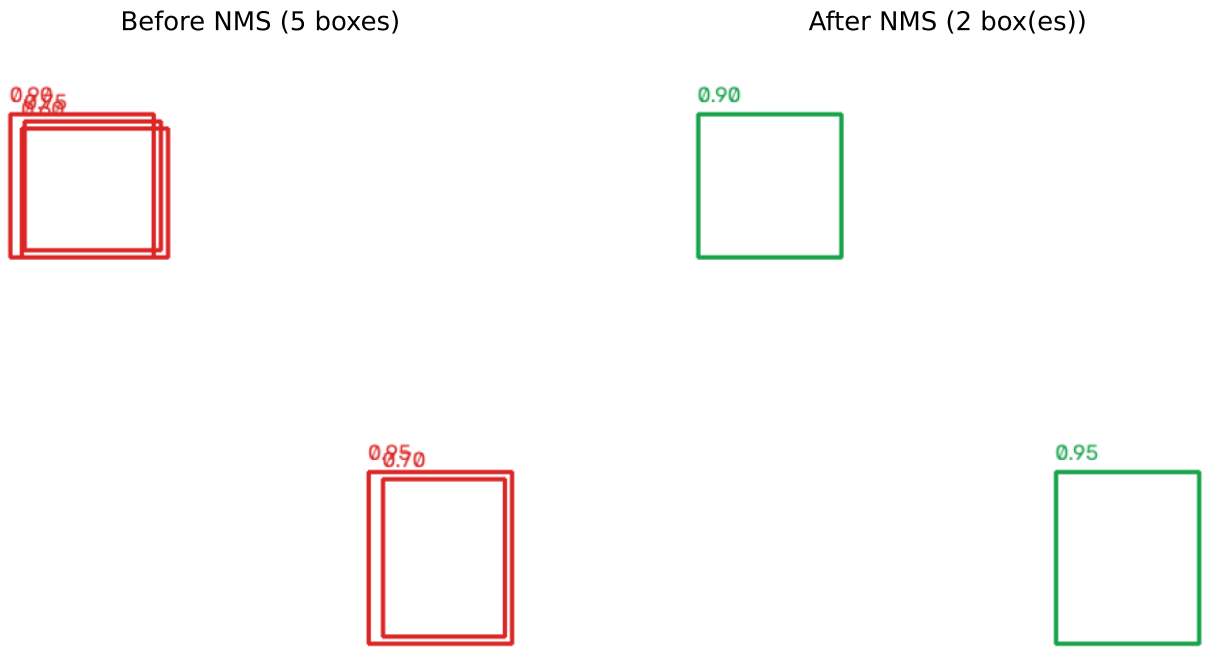


Figure 8.11: Effect of Non-Maximum Suppression (NMS): multiple redundant detections (left) are reduced to one detection per object (right).

Why It Works — Redundancy Elimination

NMS (*Non-Maximum Suppression*) does not alter the quality of detections produced by the detector. Its role is to eliminate **redundant bounding boxes** that represent the same object, keeping only the one with the highest confidence score.
Its effectiveness depends on the adopted IoU threshold. A very low value may remove boxes corresponding to different objects whose overlap is high, while a very high value may retain multiple overlapping boxes for the same object.
NMS also **does not eliminate isolated false positives**. If an incorrect region is detected by only one box, it will be kept, since there is no other redundant detection to compare it against. For this reason, NMS is applied as a **post-processing** step, using only the bounding boxes and their confidence scores produced by the detector.

8.8 Visual Segmentation: Semantic, Instance, and Panoptic

A bounding box roughly indicates the position of an object, but it does not identify which pixels belong to it. **Visual segmentation** solves this problem by assigning a label to each pixel in the image. Depending on the information produced, three main paradigms are distinguished, summarized in Table 8.1.

Table 8.1: Comparison between the main visual segmentation paradigms.

Paradigm	Question it answers	Distinguishes objects of the same class?
Semantic	"To which class does each pixel belong?"	No. All pixels of the same class receive the same label.
Instance	"Which pixels belong to each individual object?"	Yes. Each object receives its own identifier.
Panoptic	Combines the previous two.	Yes. Each pixel receives a class and, when applicable, an instance identifier.

In **semantic** segmentation, each pixel receives only the label of its class. In **instance** segmentation, in addition to the class, distinct objects belonging to the same category are differentiated from one another. **Panoptic** segmentation combines these two types of information, assigning to each pixel a class and, when applicable, an instance identifier.

8.8.1 Semantic Segmentation by Thresholding

Before the popularization of methods based on deep neural networks — the topic of **Chapter 9** —, many segmentation applications were solved using classical image processing techniques. One of the simplest examples involves separating object and background by **thresholding**, producing a binary mask. This mask characterizes a **semantic segmentation**, as each pixel now belongs to one of two classes: **coin** or **background**.

Figure 8.12 uses the coin image from the `scikit-image` library. This example revisits the concepts of thresholding and mathematical morphology studied in **Chapter 4**, replacing the previous example, which used a different coin image.

Initially, a **morphological closing** is applied to smooth small imperfections on the coin surfaces. Then, **Otsu's thresholding** produces the binary mask that separates coins and background. After removing objects connected to the image border, filling internal regions, and eliminating small noises through morphological operations, a suitable semantic mask is obtained for the subsequent steps.

8.8.2 Instance Segmentation by Labeling

A semantic mask only informs the class of each pixel, but it does not distinguish different objects belonging to the same category. To obtain an **instance segmentation**, **connected component labeling** is applied, which assigns a different identifier to each connected region of the binary mask.

In this example, each connected component corresponds to an individual coin. Thus, labeling produces an instance segmentation, allowing each coin present in the image to be identified, counted, and measured separately.

```
img_moedas = skdata.coins()

# 1. Morphological closing: smooths edges and fills small gaps on the
#    coin surface before thresholding
fechamento = mm.close(img_moedas, mm.sedisk(4))

# 2. Otsu thresholding (Chapter 4): separates coins (bright) from background (dark)
mascara_bruta = mm.threshold(fechamento)

# 3. Removes objects connected to the image border - coins cut off at the
#    edges do not form complete instances and would interfere with counting
mascara_sem_borda = mm.edgeoff(mascara_bruta)
```

```python
# 4. Hole filling: fills any internal regions not detected by the threshold,
#    ensuring each coin is a solid disk
mascara_semantica = mm.clohole(mascara_sem_borda)

# 5. Morphological opening: removes residual noise and disconnects coins that
#    may touch, preparing the mask for Labeling
mascara_limpa = mm.open(mascara_semantica, mm.sedisk(6))

# 6. Connected-component labeling: each isolated coin receives a distinct
#    instance identifier
rotulos_instancias = mm.label(mascara_limpa)

print(f"Instances (individual coins) identified: {np.max(rotulos_instancias)}")

mm.show(
    [img_moedas, mascara_bruta, mascara_sem_borda, mascara_semantica,
     mascara_limpa, rotulos_instancias],
    titles=[
        "Original",
        "1. Thresholding (Otsu)",
        "2. Border removal",
        "3. Hole filling\n(semantic mask: coin vs background)",
        "4. Opening (cleaning)",
        "5. Labeling\n(instance segmentation)"
    ],
    cols=3, figsize=(10, 6)
)
```

```
Instances (individual coins) identified: 24
```

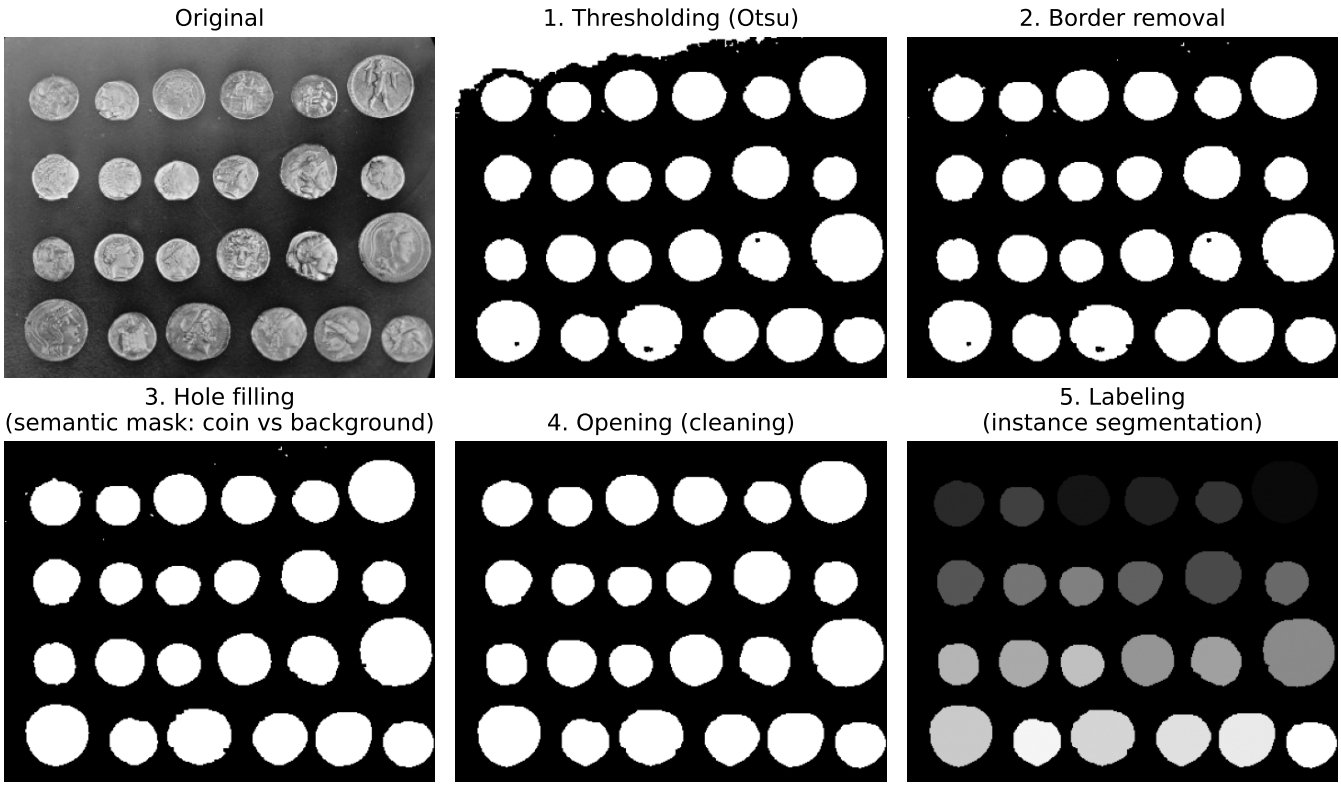


Figure 8.12: Classic segmentation (not deep learning-based): semantic mask (coin vs background) via Otsu thresholding, and instance segmentation via connected-component labeling.

🧠 Why Does It Work? — And Where the Classical Approach Encounters Limitations

In this example, segmentation is facilitated by the contrast between the coins and the background, as well as by the relative intensity homogeneity within each coin. Otsu's thresholding efficiently separates these two regions, while morphological operations remove small imperfections from the binary mask. Finally, connected-component labeling assigns a distinct identifier to each connected region, yielding instance segmentation.
This strategy, however, directly depends on the quality of the binary mask. If two objects are joined, overlapping, or exhibit insufficient contrast against the background, they may be represented as a single region or fail to be segmented correctly. Furthermore, methods based predominantly on pixel intensity have limited capacity to distinguish objects of different classes with similar appearances.
Modern segmentation methods based on deep neural networks learn visual representations directly from training data, which generally grants them a greater ability to handle variations in illumination, texture, shape, and occlusion. Panoptic segmentation extends this approach by combining, in a single representation, the semantic classification of all pixels and the individual identification of objects present in the scene.

8.8.3 Alternative: Distance Transform + *Watershed*

Morphological opening can separate touching coins by eroding the binary mask. However, this operation modifies the contour of all objects, including those that were already isolated. An alternative, introduced in **Chapter 4**, consists of combining the **distance transform** with the ***watershed*** algorithm, using markers obtained from the binary mask itself.

The procedure is carried out in three steps:

1. the **distance transform** of the binary mask is computed, assigning to each pixel of the object its distance to the nearest background;
2. a threshold is applied to the distance transform to obtain **markers** located in the central regions of the coins;
3. the ***watershed*** algorithm is used to expand these markers up to the boundaries between objects, separating coins that are in contact.

Figure 8.13 presents these steps. The markers are obtained in the central regions of the distance transform and used as seeds for the *watershed*, which propagates each label until it encounters the boundaries between neighboring objects.

Comparing with morphological opening, both approaches can separate touching coins. However, since the *watershed* uses markers to divide connected regions, without applying erosion directly to the mask, the original contours tend to be better preserved, favoring the acquisition of geometric measurements such as area, perimeter, and circularity.

```
# Reuses the semantic mask BEFORE aggressive opening (without contour erosion)
mascara_base = mascara_semantica

# 1. Distance transform: each pixel of the mask receives the distance to the nearest
background
distancia = mm.dist(mascara_base)

# Markers obtained by thresholding the distance transform.
# The central regions of the coins remain connected and are used
# as seeds for the watershed algorithm.
marcadores = mm.label(
    np.uint8(distancia > 0.5 * distancia.max())
)

# 3. Watershed: propagates each marker within the mask to the contact points between coins
```

```
rotulos_watershed = mm.watershed(marcadores, mascara_base)

print(f"Morphological opening (clean_mask): {np.max(rotulos_instancias)} instances")
print(f"Distance + watershed: {np.max(rotulos_watershed)} instances")

mm.show(
    [mascara_base, distancia, mm.dil(marcadores,mm.sedisk(5)), rotulos_watershed],
    titles=[
        "Semantic mask\n(without opening, without erosion)",
        "Distance transform",
        "Markers\n(regional maxima)",
        "Watershed\n(separated instances)"
    ],
    cols=4, figsize=(11, 3.2)
)
```

```
Morphological opening (clean_mask): 24 instances
Distance + watershed: 24 instances
```

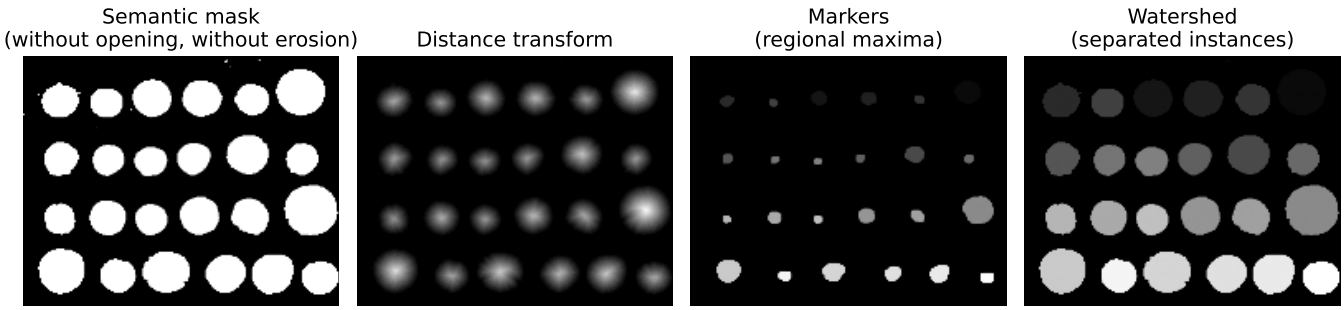


Figure 8.13: Separation of touching coins via distance transform + *watershed*, without erosion of the coin contours.

Why It Works? — Markers and *Watershed*

The distance transform assigns higher values to pixels farthest from the background, which are typically located in the most central regions of objects. By applying a threshold to this transform, markers are obtained within each coin, favoring the acquisition of one marker per object.
The ***watershed*** algorithm uses these markers as seeds and propagates their labels until two growing regions meet. The meeting points between these regions define the boundaries between adjacent objects. The performance of this method depends on the quality of the markers. Highly elongated objects, objects with irregular shapes, or those containing multiple maxima in the distance transform can generate additional markers, resulting in oversegmentation. In more complex applications, modern instance segmentation methods based on deep neural networks directly learn suitable representations to separate objects from the training data, eliminating the need for explicit marker construction and other geometric heuristics.

8.8.4 Measurement and Extraction of Attributes with `mm.measure`

After segmentation and labeling of objects, the next step involves measuring their geometric properties. To accomplish this, the `morph` library provides the `mm.measure` function, which takes a **binary** image and returns, for each object, a set of geometric descriptors organized in a list of dictionaries.

Among the computed descriptors, notable ones include the **area**, **perimeter**, **centroid**, **bounding box**, **circularity**, **solidity**, and the number of **vertices** of the approximated contour. These vertices are obtained by approximating the contour with a simplified polygon, calculated by the `approxPolyDP` function, which

preserves the general shape of the object using a reduced number of segments.

Circularity is defined as

$$\text{Circularity} = \frac{4\pi \cdot \text{Area}}{\text{Perimeter}^2},$$

assuming a value equal to 1 for a perfect circle and lower values for shapes that are progressively less circular.

Solidity is given by

$$\text{Solidity} = \frac{\text{Area}}{\text{Area of the Convex Hull}},$$

indicating how much the object fills its convex hull. Values close to 1 characterize convex objects, while lower values indicate the presence of concavities.

The number of vertices provides an indication of the complexity of the object's shape. For example, a triangle tends to produce three vertices and a rectangle four, while objects with curved contours, such as coins, usually result in polygons with a higher number of vertices, depending on the precision adopted in the approximation.

Since the coins in this example have approximately circular and convex contours, it is expected that circularity assumes high values (typically between 0.8 and 0.9) and that solidity remains very close to 1. The number of vertices depends on the parameter used in the polygonal approximation (`precision`), increasing as the approximation preserves more contour details. Together, these descriptors can be employed in object classification, identification of spurious components remaining from segmentation, and performing quantitative measurements on the image.

```
# mm.measure receives a binary image; each connected component corresponds
# to an individual coin.
medidas = mm.measure(mascara_limpa, precision=0.01)

if medidas:
    print(f"Total of cataloged objects: {len(medidas)}")
    print(f"{'ID':<4}{'Area':<10}{'Perimeter':<12}{'Circularity':<14}",end="")
    print(f" {'Solidity':<10}{'Vertices':<8}")
    for m in medidas:
        print(f"{m['id']:<4}{m['area']:<10.1f}{m['perimeter']:<12.2f}"
              f"{m['circularity']:<15.3f}{m['solidity']:<10.3f}{m['vertices']:<8}")
else:
    print("No measurements returned.")
```

```
Total of cataloged objects: 24
ID  Area      Perimeter   Circularity    Solidity  Vertices
1   1376.5    140.47      0.877          0.979     11
2   1299.5    136.47      0.877          0.978     10
3   1628.0    154.37      0.859          0.980     12
4   1866.0    162.71      0.886          0.985     11
5   2127.0    178.37      0.840          0.973     12
6   2328.0    181.68      0.886          0.983     10
7   1086.5    124.81      0.876          0.979     13
8   1053.5    121.50      0.897          0.983     9
9   1399.5    140.47      0.891          0.985     13
10  1449.0    147.05      0.842          0.972     13
11  1642.5    153.30      0.878          0.982     13
12  2987.5    207.58      0.871          0.981     13
13  1036.5    121.15      0.887          0.981     12
14  1050.5    123.15      0.870          0.979     11
15  1110.0    126.23      0.875          0.975     11
```

```
16  1135.0    127.05      0.884          0.986     11
17  1250.5    135.64      0.854          0.975     13
18  1802.0    160.71      0.877          0.982     10
19  1067.5    122.33      0.896          0.986     11
20  1154.5    129.64      0.863          0.977     10
21  1356.0    139.88      0.871          0.978     13
22  1549.5    149.30      0.874          0.982     12
23  1592.0    149.54      0.895          0.984     13
24  2509.5    187.92      0.893          0.987     14
```

It is observed that **24 coins** were identified. The solidity measures remain close to 1, indicating essentially convex contours, while circularity varies between approximately 0.84 and 0.90 due to irregularities in the discretized contour. The number of vertices ranges from 9 to 14, reflecting the polygonal approximation used to represent each coin.

8.8.5 Exporting Annotations: from `mm.measure` to the YOLO Format

In addition to geometric descriptors, `mm.measure` calculates, for each object, its **bounding box**. This information can be automatically exported using the `mm.saveMeasures` function, allowing annotation files to be generated for different applications without the need for manual labeling.

The function supports three output formats:

- `fmt="csv"`: exports all geometric descriptors, useful for exploratory analysis, measurement, and attribute-based classification;
- `fmt="txt"`: writes the descriptors in a simple tabular format;
- `fmt="yolo"`: generates annotations compatible with the format used by detectors from the **YOLO** (*You Only Look Once*) family.

In the YOLO format, each object is represented by a line containing

$$\text{class} \;\; x_c \;\; y_c \;\; w_n \;\; h_n,$$

where (x_c, y_c) represents the center of the bounding box and (w_n, h_n) its dimensions, all normalized by the image dimensions:

$$x_c = \frac{x + w/2}{W}, \qquad y_c = \frac{y + h/2}{H}, \qquad w_n = \frac{w}{W}, \qquad h_n = \frac{h}{H},$$

where (x, y) correspond to the coordinates of the top-left corner of the bounding box, (w, h) to its dimensions in pixels, and (W, H) to the width and height of the image. Normalization makes the annotations independent of image resolution, allowing the same format to be used for images of different sizes.

The following code exports the measurements extracted from the coins to the **CSV** and **YOLO** formats. The CSV output preserves all geometric descriptors calculated by `mm.measure`, while the YOLO format file contains only the class and the normalized bounding box of each object, as required by detectors from that family.

This representation will be used again in **Chapter 9**, dedicated to object detection methods based on *Deep Learning*.

```python
# Exports the measurements extracted from the coins in two annotation formats
altura_img, largura_img = img_moedas.shape[:2]

mm.saveMeasures("moedas.csv", medidas, fmt="csv")
mm.saveMeasures("moedas.txt", medidas, fmt="yolo",
                img_width=largura_img, img_height=altura_img, class_id=0)
```

```
print("--- Snippet of coins.csv ---")
with open("moedas.csv") as f:
    for linha in f.readlines()[:4]:
        print(linha.rstrip())

print("\n--- Snippet of coins.txt (YOLO format) ---")
with open("moedas.txt") as f:
    for linha in f.readlines()[:4]:
        print(linha.rstrip())
```

```
--- Snippet of coins.csv ---
id,area,perimeter,cx,cy,x,y,w,h,circularity,solidity,vertices
1,1376.5,140.47,359.57,268.85,338,250,45,41,0.877,0.979,11
2,1299.5,136.47,115.16,266.56,95,247,42,42,0.877,0.978,10
3,1628.0,154.37,245.55,264.02,222,243,49,46,0.859,0.98,12

--- Snippet of coins.txt (YOLO format) ---
0 0.938802 0.892739 0.117188 0.135314
0 0.302083 0.884488 0.109375 0.138614
0 0.641927 0.877888 0.127604 0.151815
0 0.789062 0.877888 0.130208 0.158416
```

8.8.6 Validation of Annotations by IoU: `mm.verifyBoundBox`

Before using annotations in training or model evaluation, it is important to verify whether they agree with a reference set (*ground truth*).

The `mm.verifyBoundBox` function performs this comparison using the **IoU** (*Intersection over Union*) metric, presented earlier in this chapter. For each candidate bounding box, the function calculates its overlap with the ground-truth boxes and counts those whose IoU exceeds a specified threshold.

In the following example, a **synthetic ground truth** is constructed from the first five bounding boxes obtained by `mm.measure`. Since the compared annotations are the same ones used to build the ground truth, it is expected that each object finds exactly one match with IoU $\geq 0{,}5$. The purpose is merely to illustrate the operation of the `mm.verifyBoundBox` function; in real applications, the ground truth must be obtained independently, for instance, through manual annotation.

```
# Synthetic answer matrix in the format (class, x1, y1, x2, y2), normalized
gabaritos = np.array([
    [0, *[(m["bbox"][0] + d) / largura_img for d in (0,)],
        (m["bbox"][1]) / altura_img,
        (m["bbox"][0] + m["bbox"][2]) / largura_img,
        (m["bbox"][1] + m["bbox"][3]) / altura_img]
    for m in medidas[:5]
])

acertos = 0
for m in medidas[:5]:
    correspondencias = mm.verifyBoundBox(
        object_id=0, bbox=m["bbox"], matrix=gabaritos,
        width=largura_img, height=altura_img, threshold=0.5
    )
    acertos += int(correspondencias > 0)
    print(f"Object {m['id']}: {correspondencias} answer(s) match with IoU >= 0.5")

print(f"\nTotal validated objects: {acertos}/{len(medidas[:5])}")
```

```
Object 1: 1 answer(s) match with IoU >= 0.5
Object 2: 1 answer(s) match with IoU >= 0.5
```

```
Object 3: 1 answer(s) match with IoU >= 0.5
Object 4: 1 answer(s) match with IoU >= 0.5
Object 5: 1 answer(s) match with IoU >= 0.5

Total validated objects: 5/5
```

It is observed that the five objects were correctly associated with their respective ground truth boxes, resulting in five valid matches. In real-world applications, this same strategy can be employed to automatically compare the bounding boxes produced by a detector with reference annotations, enabling a quantitative assessment of its performance through the IoU metric.

8.8.7 Viewing Saved Annotations: `mm.showBoundBox`

After exporting annotations, it is convenient to verify whether the bounding boxes were saved correctly. To this end, the `morph` library provides the `mm.showBoundBox` function, illustrated in Figure 8.14.

The function reads an annotation file in the `yolo`, `csv`, or `txt` formats, reconstructs the **bounding boxes**, and overlays them on the original image, facilitating visual inspection of the result. This makes it possible to quickly confirm whether the annotations align with the objects, without the need to examine them manually.

This function complements `mm.verifyBoundBox`. While `mm.verifyBoundBox` performs a **quantitative validation**, comparing annotations against a reference set using the IoU metric, `mm.showBoundBox` provides a **qualitative validation**, allowing for visual inspection of the reconstructed bounding boxes.

```
caminho_yolo = "moedas.txt"
img_anotada = mm.showBoundBox(img_moedas, caminho_yolo, fmt="yolo")
```

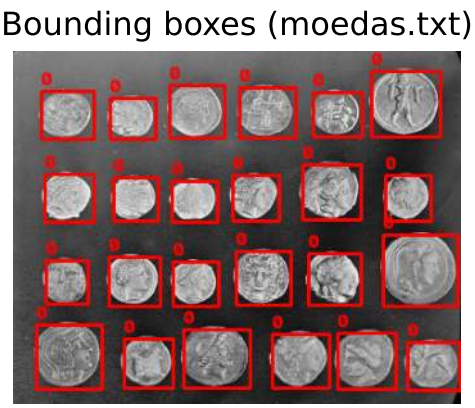


Figure 8.14: *Bounding boxes* reconstructed from the YOLO annotations file exported by *mm.saveMeasures*, overlaid on the original image of the coins.

> 🧠 Why It Works — From Segmentation to Annotations
>
> After segmenting an image, it is possible to automatically measure each object (`mm.measure`) and convert these measurements into annotations (`mm.saveMeasures`) for various formats, including the one used by YOLO-family detectors. This procedure allows for the automatic generation of initial annotation sets in controlled scenarios, significantly reducing the manual labeling workload.
> Before using these annotations in model training or evaluation, it is recommended to compare them against a reference set (*ground truth*). The function `mm.verifyBoundBox` automates this step using the **IoU** metric, enabling quantification of the agreement between the generated bounding boxes and the reference annotations. Additionally, `mm.showBoundBox` allows for visual inspection of the reconstructed annotations overlaid on the original image, facilitating the identification of normalization, positioning, or coordinate-ordering errors that may not be evident through numerical validation alone.

Together, `mm.measure`, `mm.saveMeasures`, `mm.verifyBoundBox`, and `mm.showBoundBox` implement a complete workflow for measuring segmented objects, generating annotations, evaluating them, and visually inspecting them. When segmentation yields reliable results, this workflow can significantly reduce the need for manual labeling in dataset construction.

8.9 Overview of Modern Models

The techniques presented in this chapter—such as cascade detectors, thresholding-based segmentation, and post-processing with IoU and NMS—remain important for understanding the fundamentals of Computer Vision. Currently, however, many applications employ **Deep Learning** models capable of automatically learning discriminative representations from large datasets, eliminating the need for manually defined features.

Table 8.2 presents some representative architectures for image detection and segmentation. The goal is merely to situate them within the context of the tasks studied in this chapter. Their operating principles, training, and application will be discussed in detail in **Chapter 9**.

Table 8.2: Representative Deep Learning architectures for image detection and segmentation.

Model	Task	Central Idea
YOLO (*You Only Look Once*)	Object detection	Detects objects in a single pass through the network, estimating classes and bounding boxes. Recent versions, such as **YOLO26** (2026), eliminate the *Non-Maximum Suppression* (NMS) step, making inference fully *end-to-end*.
Faster R-CNN	Object detection	Generates candidate regions and refines them before classification, prioritizing accuracy.
SSD (*Single Shot Detector*)	Object detection	Detects objects at different scales in a single pass, seeking a balance between speed and accuracy.
U-Net	Semantic segmentation	Produces a mask that classifies each pixel of the image according to its class.
Mask R-CNN	Instance segmentation	Extends Faster R-CNN by adding an individual mask for each detected object.
***Segment Anything* (SAM)**	Prompt-guided segmentation	Segments objects based on indications such as points, bounding boxes, or masks. It evolved into **SAM 2**, with support for video and temporal object tracking, and **SAM 3**, which enables segmentation guided directly by text descriptions.

One observes an evolution of the tasks addressed in this chapter. Models such as **YOLO**, **Faster R-CNN**, and **SSD** localize objects using bounding boxes. **Mask R-CNN** extends this capability by also producing a

mask for each detected instance. More recent models, such as ***Segment Anything* (SAM)**, allow segmenting objects from different types of *prompts*, making the process more flexible.

This evolution follows the sequence of concepts developed throughout the chapter: from feature matching and approximate localization via bounding boxes to the precise segmentation of pixels belonging to each object. In **Chapter 9**, these tasks will be revisited from the perspective of Deep Learning, exploring how modern neural networks automatically learn representations capable of overcoming many of the limitations of the classical methods presented here.

8.10 Limitations of Classical Approaches and Motivation for *Deep Learning*

The examples developed in this chapter illustrate three fundamental problems in Computer Vision: **feature matching**, **object detection**, and **image segmentation**. Although the classical techniques presented are efficient in several scenarios, they all share an important limitation: they rely on manually defined features to describe or identify the objects of interest.

- **ORB** uses binary descriptors capable of establishing correspondences between points under rotations and moderate scale changes, but it was not designed to recognize object categories.
- **Haar Cascade** employs a fixed set of rectangular filters, working well for objects with relatively standardized appearances, such as frontal faces, but becoming more susceptible to false positives in complex scenes.
- **Thresholding-based segmentation** exploits intensity differences between object and background, being suitable for images with good contrast, yet limited in the presence of illumination variations, textures, or scenes containing multiple object classes.

In all these cases, performance depends on the ability of the chosen features to adequately represent the variability present in the images. In real-world applications, factors such as changes in pose, illumination, scale, occlusions, and category diversity make this task increasingly difficult, reducing the generalization of classical methods.

Chapter 9 presents a different approach to these problems through **Convolutional Neural Networks (CNNs)**. Instead of using manually defined features, these models automatically learn, from large datasets, representations suitable for each task. The modern architectures presented in the previous section—such as **YOLO**, **Faster R-CNN**, **U-Net**, **Mask R-CNN**, and ***Segment Anything* (SAM)**—follow this principle and represent an evolution of the classical techniques studied in this chapter, expanding their capacity to handle the diversity and complexity of real-world images.

8.11 Summary

In this chapter, classical Computer Vision methods for **feature matching**, **object detection**, and **image segmentation** were studied. At the end, some modern architectures based on Deep Learning were presented, preparing the transition to **Chapter 9**. The main concepts covered were:

- **ORB:** keypoint detection, binary descriptor construction, and feature matching using the Hamming distance.
- **Homography and RANSAC:** robust estimation of projective transformations for image registration and removal of inconsistent matches.
- **Haar Cascade:** object detection using Haar features, integral image, and cascade classifiers.
- **Classical segmentation:** thresholding, morphological operations, connected-component labeling, distance transform, and the *watershed* algorithm for instance segmentation.
- **Object measurement and annotation:** extraction of geometric attributes with `mm.measure`, automatic annotation export (`mm.saveMeasures`), and validation via IoU (`mm.verifyBoundBox`) and visual inspection (`mm.showBoundBox`).

- **Detection evaluation:** use of the **Intersection over Union (IoU)** metric and **Non-Maximum Suppression (NMS)**.
- **Modern models:** overview of the **YOLO**, **Faster R-CNN**, **SSD**, **U-Net**, **Mask R-CNN**, and **Segment Anything (SAM)** architectures, which will serve as the foundation for studying Deep Learning methods in **Chapter 9**.

Next Steps

In this chapter, classical methods for solving problems of **feature matching**, **object detection**, and **image segmentation** were presented, based on manually defined descriptors, filters, and geometric models.

In **Chapter 9**, these same problems will be revisited from the perspective of **Deep Learning**, with an emphasis on **Convolutional Neural Networks (CNNs)**. Instead of using manually engineered features, these models automatically learn representations from large datasets, forming the foundation of the main modern Computer Vision systems for object detection, segmentation, and recognition.

8.12 🤖 Using Gemini Notebook as a Study Aid

Gemini Notebook can be used as a complementary tool to review the concepts presented in this chapter. Based on the content provided as a reference, it can answer questions, prepare summaries, clarify doubts, and explore topics interactively.

Accessing the Chapter Notebook

🚀 ACCESS GEMINI NOTEBOOK: CHAPTER 08

🌐 **Language and Programming Language**

The project for this chapter in Gemini Notebook was built using only text in **Portuguese** and code examples in **Python**. If you are studying from the English or French edition, or following the C++ track, the tutor's responses may not correspond exactly to the version you are reading.
Responses generated by Gemini Notebook are produced automatically and may contain inaccuracies. Use them as supplementary study material, cross-checking the information with the content of this book and, when necessary, with other academic sources.

8.13 Exercise List

The following exercises consolidate the concepts presented in this chapter through adaptations, experiments, and extensions of the algorithms developed throughout the text, using the didactic library `morph`.

1. **(10%)** Investigate the influence of rotations on the stability of the **ORB** detector and descriptor. Using the image `skimage.data.astronaut()`, generate rotated versions at angles of $0°, 45°, 90°, 135°, 180°$. For each case, determine the number of correspondences obtained by the `BFMatcher` and the fraction of *inliers* identified by **RANSAC**. Present the results in a table and discuss the robustness of the method to the analyzed rotations.

2. **(15%)** Compare the **FAST** corner detector with the **Harris** detector (Chapter 6). Measure the average execution time and the number of detected points on the images `skimage.data.camera()`, `skimage.data.gravel()`, and `skimage.data.brick()`. Discuss the advantages and limitations of each detector in terms of speed and repeatability.

3. **(15%)** Investigate the sensitivity of the **Haar Cascade** to the presence of noise and blur. Add Gaussian noise ($\sigma \in 5, 10, 20$) and motion blur to the image `skimage.data.astronaut()`. Vary the `scaleFactor` and `minNeighbors` parameters of the `detectMultiScale()` function and discuss the effects on false positives, false negatives, and the number of detected faces.

4. **(15%)** Manually implement the **Intersection over Union (IoU)** metric for bounding boxes and compare your results with the `mm.IoU` function. Then, create a reference box and ten boxes obtained through displacements and scale variations, building a graph that relates the applied displacement to the IoU value obtained.

5. **(15%)** Evaluate the **Non-Maximum Suppression (NMS)** algorithm. Create a synthetic scene containing three objects, each represented by at least five overlapping boxes with different confidence scores. Run the algorithm for IoU thresholds equal to $0.2, 0.4, 0.6, 0.8$, visualize the results, and discuss the influence of this parameter on the number of boxes retained.

6. **(15%)** Apply the segmentation pipeline presented in this chapter (Otsu thresholding, morphological operations, and `mm.measure`) to the image `skimage.data.coins()`. Export the measurements using `mm.saveMeasures(fmt="csv")` and produce histograms of the distributions of area, circularity, and solidity. Discuss which of these attributes are most suitable for characterizing the coins.

7. **(15%)** Construct a synthetic image containing circles, rectangles, and triangles, some partially overlapping. Perform segmentation, extract the measurements with `mm.measure`, and automatically generate annotations in YOLO format using `mm.saveMeasures(fmt="yolo")`. Then, deliberately modify some *bounding boxes* and use `mm.verifyBoundBox` to evaluate, at different IoU thresholds, how many annotations remain valid. Discuss the influence of overlap between objects on the quality of the obtained annotations.

8. **(Bonus – 10%)** Develop a simple classifier, such as **k-NN** (Chapter 7), using only the geometric attributes produced by `mm.measure` (area, circularity, solidity, and number of vertices). Use a set of synthetic shapes (circles, rectangles, and triangles) with different scales and rotations, evaluate the obtained accuracy, and discuss the capacity of these descriptors to distinguish different classes of objects.

Chapter References

The concepts and algorithms presented in this chapter were grounded in classic and contemporary references from the DIP-CV literature:

- Gonzalez (2018), for the fundamentals of image segmentation, morphological operations, connected component labeling, and geometric attribute extraction.
- Szeliski (2022), for the concepts of feature matching, geometric transformations, homographies, object detection, and segmentation in Computer Vision.
- Rublee (2011), for the description of the **ORB** algorithm (*Oriented FAST and Rotated BRIEF*), used in the detection, description, and matching of local features.
- Fischler (1981), for the original formulation of the **RANSAC** algorithm (*Random Sample Consensus*), employed in the robust estimation of geometric models in the presence of inconsistent data.
- Viola (2001), Viola (2004), and Lienhart (2002), for the foundations of the **Haar Cascade** detector, including Haar features, integral images, cascade classifiers, and extensions used in modern implementations.
- Kirillov (2019), for the definition of **panoptic segmentation**, which integrates semantic segmentation and instance segmentation into a single representation.
- Redmon (2016), Ren (2015), Ronneberger (2015), He (2016), and Kirillov (2019), for modern Deep Learning-based models applied to image detection and segmentation, including **YOLO**, **Faster R-CNN**, **U-Net**, **Mask R-CNN**, and **Segment Anything (SAM)**.

Run Colab Open GitHub

8.14 💻 Practical Part with Programming Exercises

The present list of programming exercises (PE) consolidates the theoretical formulations presented throughout Chapter 8 — Feature Matching, Object Detection, and Classical Segmentation — through an applied practical track. As in the previous chapter, the PEs isolate the **intermediate quantities** of each technique — the distance between binary descriptors, the terms of an integral image, the count of *inliers* of a candidate model, the overlap between bounding boxes, and the label of each connected component — allowing each step of the reasoning to be manually validated without relying on OpenCV or external images.

The sequencing of the exercises reproduces the conceptual flow of the chapter: it begins with the **Hamming distance**, the core of binary descriptor matching such as ORB; it advances to the counting of ***inliers*** that underpins **RANSAC** in the robust estimation of a homography; it proceeds with the **integral image**, the computational trick that makes ***Haar Cascade*** viable in real time; it delves into **IoU and Non-Maximum Suppression**, the post-processing common to virtually every object detector; and it concludes with **connected component labeling**, the classical approach — and its limitations — for segmenting individual instances in a binary mask.

🎯 Objective of this Notebook

This notebook allows you to develop, validate, organize, and test solutions for **Programming Exercises (EPs)** in interactive environments, such as Colab, using the same test cases as Moodle, and copying them there only when registering the official grade.

Download

Download `morph.py` and `testsuite.py` by running the cell below:

```
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup(testsuite=True)
from morph import mm
from testsuite import TestSuite
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Running the Tests

To evaluate the tests, run `TestSuite("EP08_01.extension").run()` in a new cell, replacing the extension with that of the language used (`.py`, `.java`, `.c`, `.cpp`, `.js`, or `.r`). The system downloads the test cases from GitHub, runs the program, and calculates the grade automatically.

To test Python code directly, without saving a file, use `run_code(code)` passing the code as a *string* in a variable `code`:

```
code = """
# ... your code here ...
"""
TestSuite("EP08_01").run_code(code)
```

Summary of Methods in `morph.py` (Ch. 8)

The `morph.py` library provides functions for connected component analysis, contour extraction, geometric metrics, and annotations:

1. **Components and Contours (`connectedComponents`, `findContours`)** Label connected regions and extract contours from binary images.
2. **Contour Properties (`contourArea`, `arcLength`, `convexHull`, `approxPolyDP`, `fitLine`)** Compute area, perimeter, convex hull, polygonal approximation, and line fitting for a contour.
3. **Enclosing Geometry (`boundingRect`, `minAreaRect`, `boxPoints`, `minEnclosingCircle`, `fitEllipse`)** Determine enclosing rectangles (aligned or oriented), ellipses, and the smallest bounding circle.
4. **Measurement Extraction and Persistence (`measure`, `saveMeasures`)** Extract geometric descriptors of objects (area, circularity, solidity, centroid) and export the data to CSV, text, or YOLO format.
5. **Evaluation and Visualization (`IoU`, `verifyBoundBox`, `showBoundBox`)** Compute the Intersection over Union, validate bounding boxes against ground truths, and draw annotated bounding boxes on the image.

8.14.1 EP08_01 🟢 Hamming Distance and Binary Descriptor Matching

ORB, used in Practical Project 1 of this chapter, describes the neighborhood of each keypoint as a sequence of bits — and, therefore, the comparison between two descriptors does not use the Euclidean distance of the k-NN from Chapter 7, but rather the **Hamming distance**: the number of positions in which the bits differ. Before calling `cv2.BFMatcher(cv2.NORM_HAMMING)`, you were tasked with manually implementing this brute-force matching — the same step that, when executed internally by OpenCV, precedes the robust homography estimation by RANSAC.

8.14.1.1 📋 Implementation Guidelines

1. **Quantities:** Read the integers N and M — the number of descriptors extracted from image A and image B, respectively.
2. **Descriptors from A:** Read N lines, each containing a binary descriptor (a *string* of characters `0` and `1`, all of the same length).
3. **Descriptors from B:** Read M lines, in the same format.
4. **Threshold:** Read the integer τ — the maximum acceptable Hamming distance for a match to be considered valid.
5. **Hamming Distance:** For two binary descriptors a and b of the same length,

$$d_H(a, b) = \sum_k \mathbb{1}[a_k \neq b_k],$$

   i.e., the count of positions in which the bits differ.
6. **Nearest-neighbor matching:** For each descriptor a_i from A (i in reading order, starting at 0), compute its Hamming distance to **all** descriptors from B and find the one with the smallest distance. In case of a tie between two or more descriptors from B with the same minimum distance, choose the one with the **smallest index**.
7. **Threshold filtering:** If the smallest distance found is $\leq \tau$, the match is valid; otherwise, a_i has no corresponding match.
8. **Output:** For each i from 0 to $N-1$, in reading order, print a line: `i j d` if there is a valid match (where j is the index of the chosen descriptor from B and d its distance), or `i -1` otherwise. At the end, print `Total correspondências válidas: X`.

8.14.1.2 📌 Computational Constraints

- **Same length:** all descriptors (from A and B) have exactly the same number of bits.
- **Brute force:** compare each descriptor from A to **all** those from B — no indexing or acceleration structure is required.
- **Tie-breaking by smallest index in B**, and **never** by the reading order of A (which is already natural, since each a_i is handled independently).

8.14.1.3 🧠 Theoretical Foundation

Element	Role in ORB matching
Binary descriptor (BRIEF)	Each bit is the result of an intensity comparison between two pixels in the neighborhood
Hamming distance	Dissimilarity metric between binary *strings*; much faster to compute than the Euclidean distance (XOR operation + bit counting)
Nearest neighbor	Matching criterion: each point from A is paired with the point from B whose descriptor is most similar
Threshold τ	Filters unreliable matches even before RANSAC — but, as discussed in the chapter, some incorrect matches still pass, requiring the robustness of RANSAC

8.14.1.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integers N and M.
- Next N lines: one binary descriptor per line (*string* of `0`s and `1`s).
- Next M lines: one binary descriptor per line, in the same format.
- Last line: Integer τ.

Output:

- N lines, one per descriptor from A, in the format `i j d` or `i -1`.
- Last line: `Total correspondências válidas: X`.

8.14.1.5 📌 Examples

Input	Output	Observation
3 3 10101010 11110000 00001111 10101011 00001110 11111111 2	0 0 1 1 -1 2 1 1 Total correspondências válidas: 2	The descriptor `11110000` has no match: its nearest neighbor is at distance 4, above the threshold $\tau = 2$.

```
%%writefile EP08_01.py
# Python code
```

```
Overwriting EP08_01.py
```

```
TestSuite("EP08_01.py").run()
```

```
<IPython.core.display.HTML object>
```

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-ep0801.html

Figure 8.15: EP08_01 Simulator: Hamming Distance Between Two Binary Descriptors

8.14.2 EP08_02 🟢 Homography and RANSAC: Voting by *Inliers*

RANSAC, introduced in the section "Mathematical Modeling: Homography and RANSAC," repeats a cycle of three steps — sampling a minimal set, estimating a candidate model, and counting how many correspondences are consistent with it (the ***inliers***) — keeping at the end the most voted model. The model estimation step from 4 points (step 2) involves linear algebra that is beyond the scope of this assignment; here, you are directly given a set of **already candidate** homographies — as if each one had been estimated from a different random sample — and you are tasked with reproducing exactly the algorithm's decisive step: **apply each model to all correspondences and count its *inliers***, choosing the winner.

8.14.2.1 📋 Implementation Guidelines

1. **Correspondences:** Read the integer N and then N lines with four real numbers each, $x\ y\ x'\ y'$ — a point from image A and its (possibly incorrect) corresponding point in image B, exactly as produced by the matching step of EP08_01.
2. **Candidate models:** Read the integer K (number of candidate homographies) and the real number ε (reprojection error threshold). Then, read K lines, each with nine real numbers $h_{11}\ h_{12}\ h_{13}\ h_{21}\ h_{22}\ h_{23}\ h_{31}\ h_{32}\ h_{33}$ — the elements of the candidate matrix H, in row-major reading order.
3. **Reprojection:** For each correspondence (x, y, x', y') and each candidate model H_k, compute the projected point
$$\begin{bmatrix}\hat{x}\\ \hat{y}\\ \hat{w}\end{bmatrix} = H_k \begin{bmatrix}x\\ y\\ 1\end{bmatrix}, \qquad (\hat{x}/\hat{w},\ \hat{y}/\hat{w}) \text{ is the projected point.}$$
4. **Reprojection error:** $e = \sqrt{(\hat{x}/\hat{w} - x')^2 + (\hat{y}/\hat{w} - y')^2}$.
5. **Inlier counting:** A correspondence is an *inlier* of model H_k if $e \le \varepsilon$.
6. **Best model selection:** The winning model is the one with the largest number of *inliers*; in case of a tie, choose the one with the **smallest index** k (the first one encountered during the RANSAC iterative cycle).
7. **Output:** For each model k from 0 to $K-1$, in reading order, print `Modelo k: I inliers`. Finally, print `Melhor modelo: k_best com I_best inliers`.

8.14.2.2 📌 Computational Constraints

- **Inclusive comparison:** a reprojection error **exactly equal** to ε counts as an *inlier* ($e \le \varepsilon$).
- **No estimation of H:** the matrices are already provided ready-made — there is no need (nor expectation) to solve any linear system.
- **Tie resolved by the smallest index**, reflecting the natural behavior of an iterative algorithm that traverses models in order and only replaces the best found so far when a new model **strictly surpasses** it.

8.14.2.3 🧠 Theoretical Foundation

Element	Role in RANSAC
Minimal sample (4 pairs)	Sufficient to determine the 8 degrees of freedom of a homography
Candidate model H_k	Estimated from a minimal sample; can be good or bad, depending on whether the sample contained *outliers*
Reprojection error	Measures how well the model "predicts" each observed correspondence
Inlier vs. *outlier*	Correspondences consistent with the winning model (*inliers*) vs. the remaining ones, typically incorrect matches from the *matching* step
Final refinement	In practice, after selecting the best model, RANSAC recomputes it using **only** its *inliers* — a step not required in this assignment

8.14.2.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer N.
- Next N lines: four real numbers $x\ y\ x'\ y'$.
- Next line: Integer K and real number ε.
- Next K lines: nine real numbers (elements of H_k, row-major).

Output:

- K lines in the format `Modelo k: I inliers`.
- Last line: `Melhor modelo: k_best com I_best inliers`.

8.14.2.5 📌 Examples

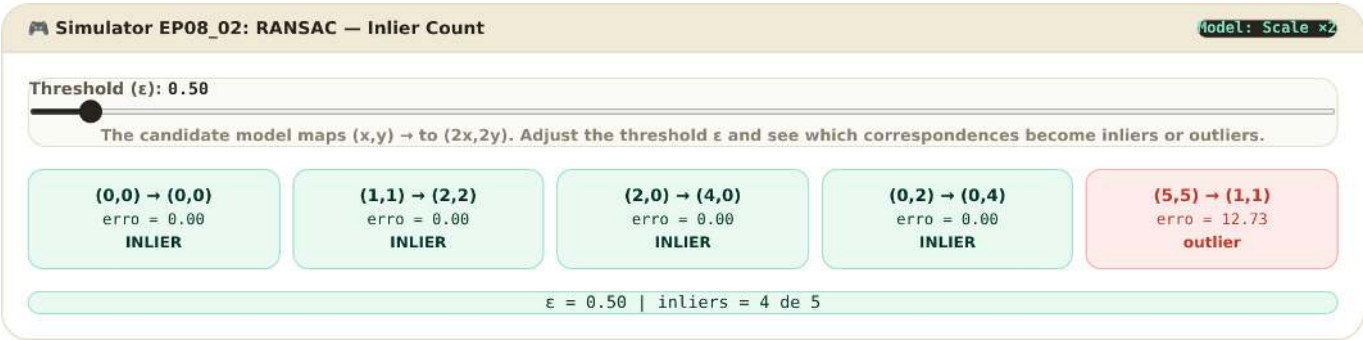
Input	Output	Observation
5 0 0 0 0 1 1 2 2 2 0 4 0 0 2 0 4 5 5 1 1 2 0.5 2 0 0 0 2 0 0 0 1 1 0 0 0 1 0 0 0 1	Modelo 0: 4 inliers Modelo 1: 1 inliers Melhor modelo: 0 com 4 inliers	Model 0 (scale $\times 2$) correctly explains 4 of the 5 correspondences; the 5th, $(5,5) \to (1,1)$, is an *outlier* that neither model explains well.

```
%%writefile EP08_02.py
# Python code
```

```
Overwriting EP08_02.py
```

```
TestSuite("EP08_02.py").run()
```

```
<IPython.core.display.HTML object>
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-ep0802.html

Figure 8.16: EP08_02 Simulator: RANSAC — Voting by Inliers among Candidate Models

8.14.3 EP08_03 🟢 Integral Image: Rectangular Sums in Constant Time

Imagine a security camera processing 30 frames per second, and for each frame the system must scan the image at dozens of positions and scales, testing at each one a set of rectangular features to decide "is there a face here?". If computing the sum of intensities for each rectangle required summing pixel by pixel, the system would have no chance of real-time evaluation — the bottleneck would lie precisely in the most repetitive part of the algorithm. It is exactly this bottleneck that the integral image eliminates.

The Haar Cascade evaluates thousands of rectangular features per window, at multiple positions and scales — something unfeasible in real time if each rectangle required summing its pixels one by one. The **integral image**, defined in the section on Haar Cascade, solves this problem: once precomputed, the sum of intensities of **any** rectangular region is obtained with just four lookups and three arithmetic operations, regardless of the size of the rectangle.

You have been tasked with implementing this structure from scratch: first, compute the integral image from the original image; then, answer arbitrary rectangular queries using it.

8.14.3.1 📋 Implementation Guidelines

1. **Input:** Read the dimensions $H \times W$ of the image and its $H \times W$ integer intensity values.
2. **Integral image:** Compute, for each position (i, j) (indexing starting from 0, `[row][column]`),

$$II(i,j) = \sum_{i' \le i,\ j' \le j} I(i', j'),$$

   that is, the sum of all pixels above and to the left of (i, j), including the position itself.
3. **Queries:** Read the integer Q and then Q lines, each with four integers $x_1\ y_1\ x_2\ y_2$ — the top-left and bottom-right corners of a rectangle, **both inclusive**, with $0 \le x_1 \le x_2 < W$ and $0 \le y_1 \le y_2 < H$.
4. **Rectangular sum in O(1):** For each query, compute the sum of intensities within the rectangle using exclusively values already present in II (without traversing the original pixels):

$$S(x_1, y_1, x_2, y_2) = II(y_2, x_2) - II(y_2, x_1{-}1) - II(y_1{-}1, x_2) + II(y_1{-}1, x_1{-}1),$$

   treating any term with a row or column index equal to -1 as 0.
5. **Output:** First, print the complete integral image — H lines with W integers each. Then, for each query, print a single integer: the sum of the corresponding region.

8.14.3.2 📌 Computational Constraints

- **Do not recalculate by brute force:** the answer to each query must use the four-term formula on II, not a direct sum of the rectangle's pixels (even though the numerical result is the same, the point of the exercise is precisely this technique).
- **Rectangles with inclusive coordinates:** (x_1, y_1) and (x_2, y_2) belong to the summed region.
- **Boundary handling:** when querying II with index -1 (when $x_1 = 0$ or $y_1 = 0$), use the value 0.

8.14.3.3 🧠 Theoretical Foundation

Element	Role in Haar Cascade
Integral image II	Precomputed once per image, in time $O(HW)$
O(1) query	Each Haar feature (difference between sums of rectangular regions) is evaluated with few operations, regardless of the rectangle's area
Scalability	It is this constancy that makes it feasible to evaluate thousands of features, at multiple positions and scales, in real time
Inclusion-exclusion principle	The four terms of the formula sum the desired region and subtract exactly the areas counted in excess

8.14.3.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integers H and W.
- Next H lines: W integers each (original image).
- Next line: Integer Q.
- Next Q lines: four integers x_1 y_1 x_2 y_2.

Output:

- H lines with W integers each (the integral image).
- Q lines, one per query, with the sum of the corresponding region.

8.14.3.5 📌 Examples

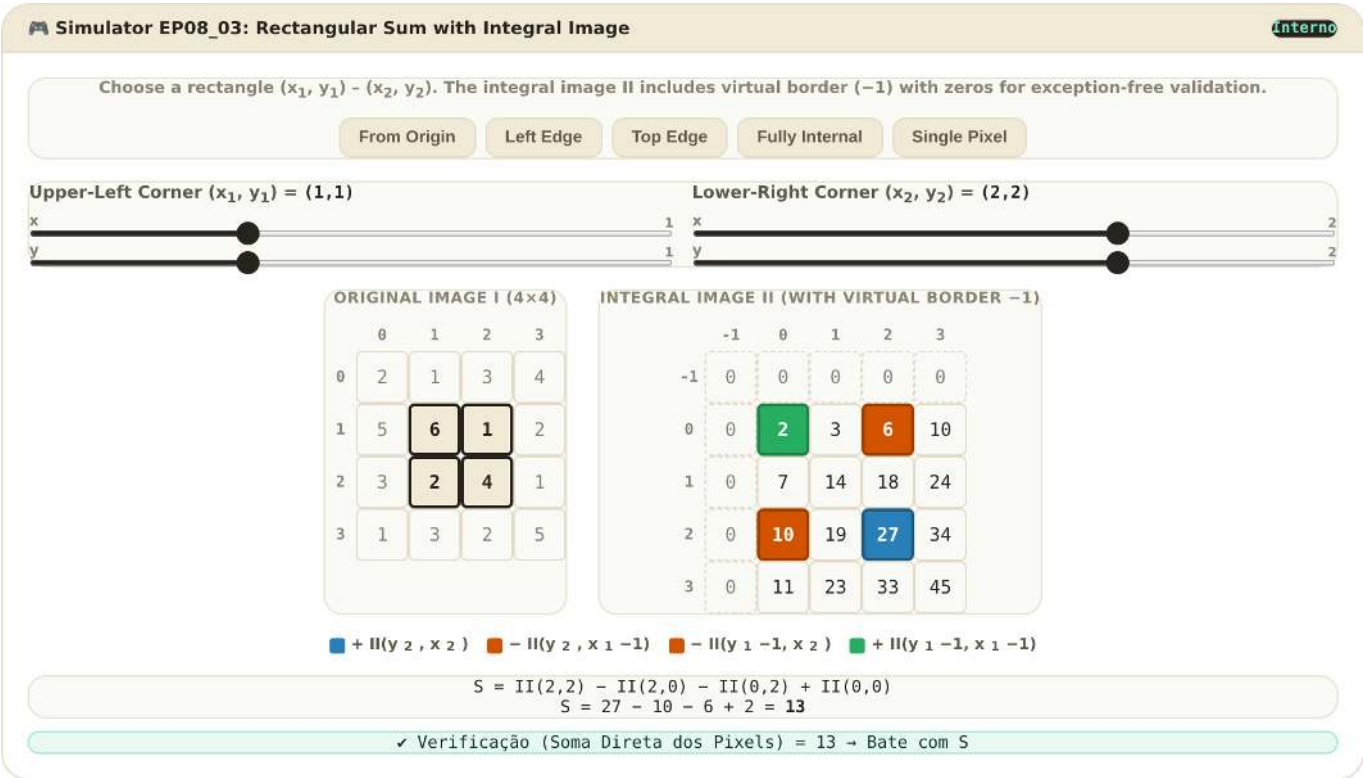
Input	Output	Observation
3 3 1 2 3 4 5 6 7 8 9 1 0 0 2 2	1 3 6 5 12 21 12 27 45 45	The query covers the entire image; the sum coincides with $II(2,2)$ and with the sum of all 9 values.
3 3 1 2 3 4 5 6 7 8 9 2 1 1 2 2 0 0 1 1	1 3 6 5 12 21 12 27 45 28 12	The first query uses the four terms of the formula; the second coincides directly with $II(1,1)$, since it starts at the origin.

```
%%writefile EP08_03.py
# Python code
```

```
Overwriting EP08_03.py
```

```
TestSuite("EP08_03.py").run()
```

```
<IPython.core.display.HTML object>
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-ep0803.html

Figure 8.17: EP08_03 Simulator: Rectangular Sum in O(1) — Multiple Edge Cases

8.14.4 EP08_04 🟢 IoU and Non-Maximum Suppression (NMS)

The figure in this section showed the effect of Non-Maximum Suppression on a set of boxes produced by a *sliding window* detector: multiple redundant detections per object were reduced to a single box per object. You were tasked with reimplementing, byte by byte, the two functions that produced that result — `calcular_iou` and `supressao_nao_maximos` — to confirm, with your own hands, exactly the numbers presented in the chapter.

8.14.4.1 📋 Implementation Guidelines

1. **Input:** Read the integer N (number of boxes) and the real τ (IoU threshold). Then read N lines, each with five reals x_{min} y_{min} x_{max} y_{max} score.
2. **Intersection over Union:** For two boxes A and B,

$$\mathrm{IoU}(A, B) = \frac{\mathrm{area}(A \cap B)}{\mathrm{area}(A \cup B)},$$

   with the intersection area being zero when the boxes do not overlap.
3. **NMS Algorithm** (exactly as described in the chapter):
   a. Sort the boxes by `score` in descending order (ties preserve the original reading order).
   b. Select the highest-scoring box among the remaining ones; add it to the output and remove it from the list.
   c. Discard from the remaining list **all** boxes whose IoU with the selected box is **greater than or equal** to τ — only boxes with IoU $< \tau$ remain as candidates.
   d. Repeat steps (b)–(c) until the list of remaining boxes is empty.
4. **Output:** For each retained box, in the order it was selected, print its original index (reading position, starting from 0) and its `score`, with 2 decimal places. At the end, print `Total mantidas: X`.

8.14.4.2 📌 Computational Constraints

- **Attention to the direction of the threshold:** contrary to what one might assume, a box is **suppressed** when IoU $\geq \tau$ (not only when IoU $> \tau$) — follow exactly this criterion, the same as the chapter's reference code.
- **Original indices:** the output references the reading position of each box in the input, not its position after sorting by `score`.
- **Area without the +1 pixel adjustment:** use area $= (x_{max} - x_{min}) \times (y_{max} - y_{min})$, exactly as in the chapter (without the "+1" adjustment sometimes used in other conventions).

8.14.4.3 🧠 Theoretical Foundation

Element	Role in Post-Processing
IoU	Quantifies the spatial overlap between two bounding boxes
Sliding window (Haar Cascade)	Typically produces multiple overlapping detections for the same object, at nearby positions and scales
Threshold τ	Controls the aggressiveness of suppression: too low merges nearby objects; too high lets redundancies pass
Sorting by `score`	Ensures that, among redundant boxes, the one with the highest confidence always survives

8.14.4.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer N and real τ.
- Next N lines: five reals x_{min} y_{min} x_{max} y_{max} score.

Output:

- One line per retained box, in selection order: `index score` (score with 2 decimal places).
- Last line: `Total mantidas: X`.

8.14.4.5 📌 Examples

Input	Output	Observation
5 0.4 50 50 150 150 0.90 60 55 155 145 0.75 58 60 160 150 0.60 300 300 400 420 0.95 310 305 395 415 0.70	3 0.95 0 0.90 Total mantidas: 2	Exactly the example from the chapter's figure: 5 redundant boxes (2 objects) become 2 final detections. The IoU between the 1st and 2nd boxes is ≈ 0.775, well above $\tau = 0.4$.

```
%%writefile EP08_04.py
# Python code
```

```
Overwriting EP08_04.py
```

```
TestSuite("EP08_04.py").run()
```

```
<IPython.core.display.HTML object>
```

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-ep0804.html

Figure 8.18: EP08_04 Simulator: IoU and Non-Maximum Suppression

8.14.5 EP08_05 🟡 Connected Component Labeling: Instance Segmentation

The classic segmentation example in this chapter separated coin "instances" simply by their spatial disconnection in the binary mask resulting from Otsu's thresholding. That final step—labeling each connected component with an instance identifier—is exactly what you have been tasked with implementing here, from scratch, on an already prepared binary mask (0 = background, 1 = object), as if it were a manual reimplementation of `cv2.connectedComponents`.

This exercise also exposes, in a very concrete way, the limitation discussed in the chapter: the result depends entirely on how "neighborhood" between pixels is defined—and, as you will see in the second example, two diagonal pixels can be considered the same instance or different instances, depending solely on the chosen **connectivity**, not on any semantic notion of an object.

8.14.5.1 📋 Implementation Guidelines

1. **Input:** Read the dimensions $H \times W$ of the binary mask and its $H \times W$ values (0 or 1).
2. **Connectivity:** Read the integer $c \in \{4, 8\}$. Under 4-connectivity, the neighbors of (i, j) are $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, and $(i, j+1)$. Under 8-connectivity, the four diagonals are added: $(i-1, j-1)$, $(i-1, j+1)$, $(i+1, j-1)$, and $(i+1, j+1)$.
3. **Component discovery:** Traversing the mask in a row-by-row sweep, from left to right and top to bottom, whenever a pixel with value 1 that is still unlabeled is found, it starts a **new component**: assign to it the next available label (the first discovered component receives label 1, the second label 2, and so on) and propagate this same label to all pixels with value 1 reachable from it through a chain of neighbors (according to the chosen connectivity)—by breadth-first search, depth-first search, or *union-find*, at your discretion.
4. **Background pixels:** remain with label 0 and do not belong to any instance.
5. **Output:** First, print the complete label map—H lines with W integers each. Then, for each label ℓ from 1 to K (in discovery order), print `Instance l: A pixels`, where A is the number of pixels with that label. Finally, print `Total instances: K`.

8.14.5.2 📌 Computational Constraints

- **Discovery order = sweep order:** labels are numbered in the order in which each new component is found by the row-wise sweep, not by size or position.
- **Explicit connectivity:** two pixels with value 1 belong to the same instance only if there exists a chain of neighbors **according to** c linking one to the other—do not mistakenly use the opposite connectivity.
- **Pure binary mask:** all input values are exactly 0 or 1.

8.14.5.3 🧠 Theoretical Background

Element	Role in classic instance segmentation
Thresholding (Otsu, Chap. 4)	Previous stage that produces the binary mask from the intensity image
Connected component	Each instance is defined **solely** by the spatial connectivity of object pixels, without any notion of shape, class, or appearance
4- vs. 8-connectivity	Parameter that alters the result: under 8-connectivity, two blobs joined only diagonally become a single instance
Central limitation	The technique merges instances that touch or overlap (even if they are clearly distinct objects), because there is no notion of "object"—only of "connected region"

8.14.5.4 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integers H and W.
- Next H lines: W integers (0 or 1) each.
- Last line: Integer c (4 or 8).

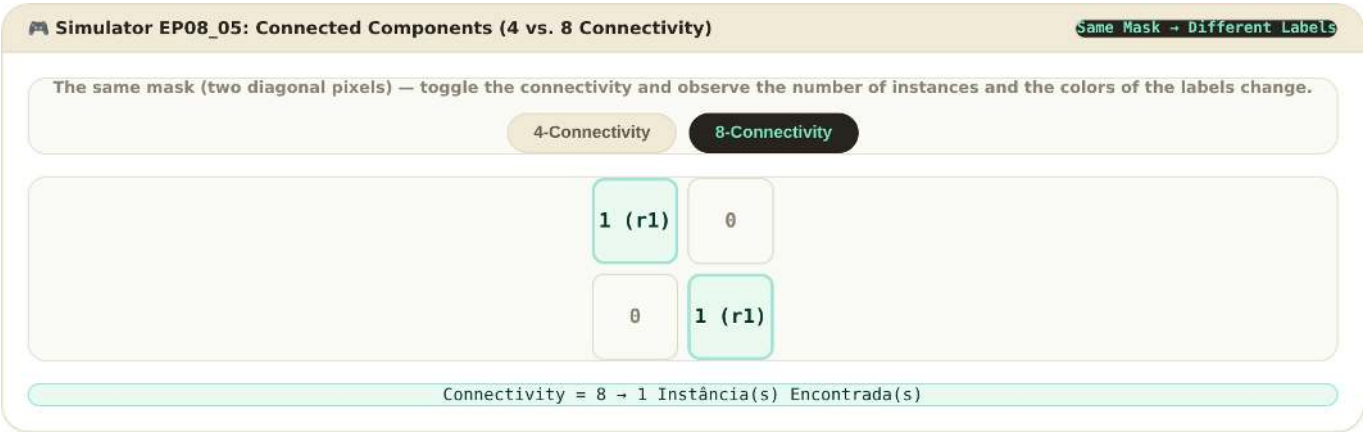
Output:

- H lines with W integers each (the label map).
- One line per instance, in discovery order: `Instance l: A pixels`.
- Last line: `Total instances: K`.

8.14.5.5 📌 Examples

Input	Output	Observation
6 6 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 8	0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 0 0 0 0 2 2 Instance 1: 4 pixels Instance 2: 4 pixels Total instances: 2	Two clearly separated 2×2 blocks: the result is the same under 4- or 8-connectivity.
2 2 1 0 0 1 8	1 0 0 1 Instance 1: 2 pixels Total instances: 1	Under 8-connectivity, the two diagonal pixels belong to the **same** instance. Repeat this example with $c = 4$: the result changes to 2 instances of 1 pixel each—purely due to the change in connectivity, with no difference in the mask.

```
%%writefile EP08_05.py
# Python code
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-ep0805.html

Figure 8.19: Simulator EP08_05: Connected Components Labeling — 4 vs. 8 Connectivity

```
Overwriting EP08_05.py
```

```
TestSuite("EP08_05.py").run()
```

```
<IPython.core.display.HTML object>
```

8.14.6 EP08_06 🟡 *Bounding Boxes*, Centroids, and Instance Properties with `mm.measure`

In the previous exercise (**EP08_05**), you can observe how segmentation by connected components labels contiguous binary regions to separate instances. However, for detection, tracking, and quantitative object analysis tasks, the simple label map is not sufficient. It becomes necessary to extract **spatial and geometric metrics** that characterize each instance individually.

This EP focuses on calculating and automatically extracting the fundamental computer vision properties for each connected component found in the binary mask, using the native `mm.measure(img)` method from the `morph` library:

1. **Bounding Box:** The smallest axis-aligned rectangle that completely encloses the instance, defined by its top-left corner (x, y), width w, and height h.
2. **Geometric Centroid $(\bar{x}, \bar{y})$:** The center of mass of the instance on the discrete grid, equivalent to the first-order spatial moments M_{10}/M_{00} and M_{01}/M_{00}.
3. **Geometric Contour Area (A):** The area enclosed by the instance contour, calculated via `mm.contourArea(c)`.

8.14.6.1 📋 Implementation Guidelines

1. **Input:** Read the dimensions $H \times W$ of the binary mask, the $H \times W$ values (0 or 1), and the connectivity parameter $c \in \{4, 8\}$.
2. **Automatic Extraction with `mm.measure`:** Pass the binarized image to the `mm.measure(img_bin)` function, which extracts OpenCV contours and returns a list of dictionaries containing the geometric properties of each instance.
3. **Returned Properties:** For each dictionary m in the list returned by `medidas = mm.measure(img_bin)`:
   - **Area (`area`):** Numerical value of the geometric contour area `mm.contourArea(c)`.
   - **Bounding Box (`bbox`):** Tuple (x, y, w, h) representing the top-left corner, width, and height.
   - **Centroid (`center`):** Tuple (c_x, c_y) with the coordinates of the center of mass M_{10}/M_{00} and M_{01}/M_{00}. Format with **two decimal places**.
4. **Output:** For each instance $1, \ldots, K$ found (ordered by discovery/position in the image), print a line containing its properties. Finally, print the total number of instances.
   - For sorting, use `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`.

8.14.6.2 🧠 Theoretical Foundation

Property in `mm.measure`	Mathematical Calculation / Discrete Logic	Practical Application in Vision
`bbox` (OpenCV)	$[x, y, w, h] = [\min(c), \min(r), \Delta c + 1, \Delta r + 1]$	Classic OpenCV format. *Note: networks such as YOLO convert this rectangle to* (c_x, c_y, w, h) *normalized.*
`center`	$\bar{x} = \frac{M_{10}}{M_{00}}, \quad \bar{y} = \frac{M_{01}}{M_{00}}$	Exact center of mass of the mask (used in tracking and trajectory analysis).
`area`	$A = \text{contourArea}(C)$ (Polygon Formula)	Continuous metric of the object's surface.

8.14.6.3 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integers H and W.
- Next H lines: W integers (0 or 1) each.
- Last line: Integer c (4 or 8).

Output:

- One line per instance in discovery order: `Instance l: Area=A, BBox=(x,y,w,h), Centroid=(cx,cy)`
- Last line: `Total instances: K`.

8.14.6.4 📌 Examples

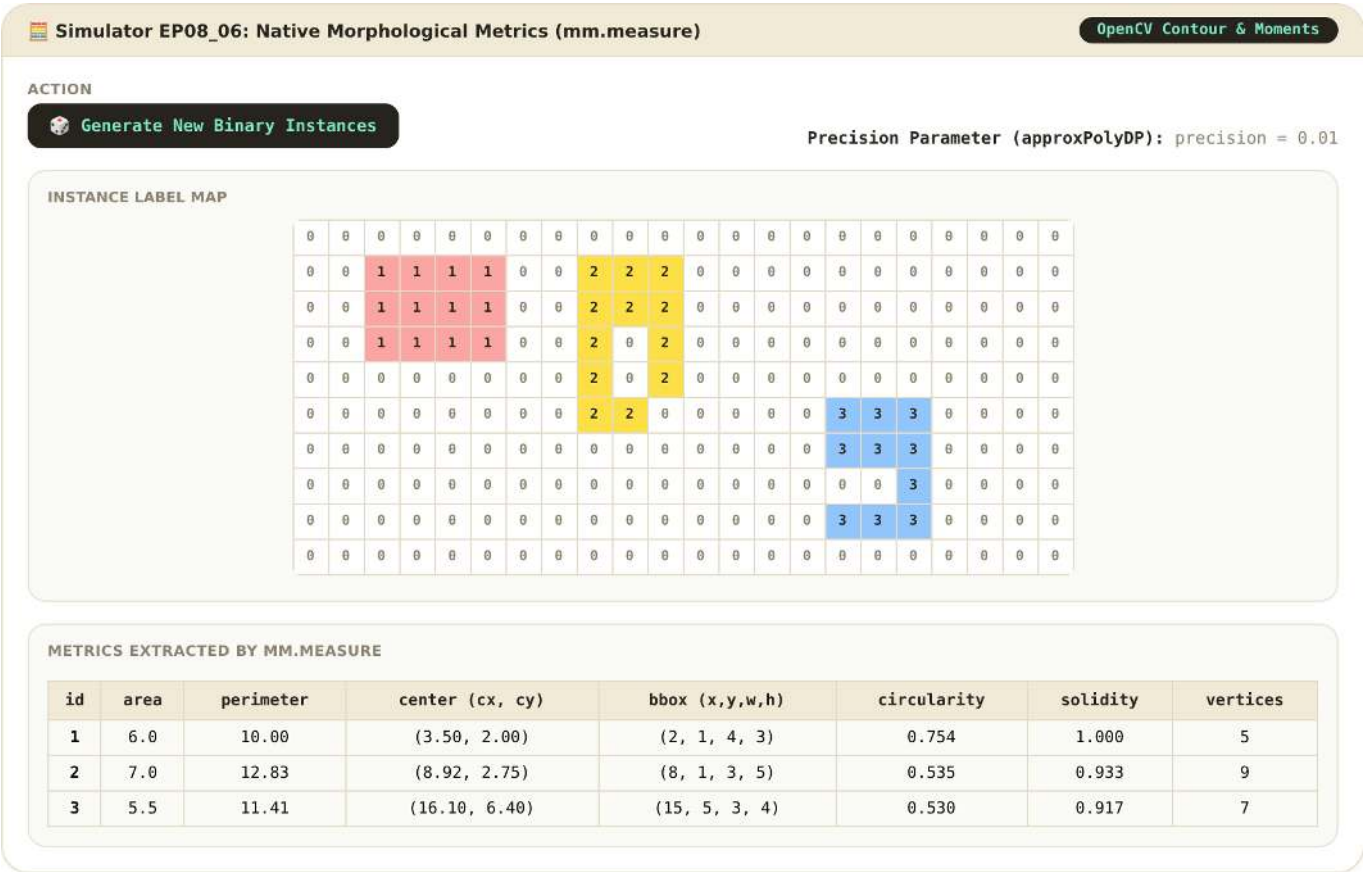
Input	Output	Observation
6 6 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 8	Instance 1: Area=1.0, BBox=(1,1,2,2), Centroid=(1.50,1.50) Instance 2: Area=1.0, BBox=(4,4,2,2), Centroid=(4.50,4.50) Total instances: 2	Aligned 2×2 blocks. The geometric contour area calculation results in 1.0. The centroid of the block in columns 1–2 and rows 1–2 is exactly (1.50, 1.50).
4 6 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 4	Instance 1: Area=2.0, BBox=(1,1,4,2), Centroid=(2.40,1.20) Total instances: 1	Asymmetric inverted "T" shaped object. The geometric contour area is 2.0. The centroid reflects the pixel distribution of the object.

```
%%writefile EP08_06.py
# Python code
```

```
Overwriting EP08_06.py
```

```
TestSuite("EP08_06.py").run()
```

```
<IPython.core.display.HTML object>
```



id	area	perimeter	center (cx, cy)	bbox (x,y,w,h)	circularity	solidity	vertices
1	6.0	10.00	(3.50, 2.00)	(2, 1, 4, 3)	0.754	1.000	5
2	7.0	12.83	(8.92, 2.75)	(8, 1, 3, 5)	0.535	0.933	9
3	5.5	11.41	(16.10, 6.40)	(15, 5, 3, 4)	0.530	0.917	7

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-ep0806.html

Figure 8.20: Simulador EP08_06: Extraction of *Bounding Boxes*, Centroids and Properties with *mm.measure*

8.14.7 EP08_07 🟡 Salt-and-Pepper Noise Removal and Object Measurement

In this exercise, you will apply morphological filtering to clean a binary image corrupted by **salt-and-pepper** noise (isolated pixels of value `1` in the background and `0` inside objects). After cleaning, the program must extract the geometric measurements of the remaining connected components, sort them, and display the final metrics table.

8.14.7.1 📋 Implementation Guidelines

1. **Input:** read two integers H and W (image height and width) from the first line, followed by H lines containing the binary matrix with pixels `0` and `1` separated by spaces.
2. **Morphological Filtering:** apply a chain of **Opening** (to eliminate salt noise in the background) followed by **Closing** (to fill pepper noise inside objects) using a 3×3 structuring element.
3. **Printing the Cleaned Image:** print the resulting matrix with values `0` and `1` separated by spaces.
4. **Geometric Measurements:** for each object identified in the cleaned matrix, extract:

- `id`: sequential numeric identifier (reassigned after sorting);
- `area`: area calculated via contour (`cv2.contourArea`);
- `perimeter`: contour perimeter (`cv2.arcLength`);
- `cx`, `cy`: center of mass (centroid via `cv2.moments`);
- `x`, `y`, `w`, `h`: coordinates of the bounding rectangle (`cv2.boundingRect`);
- `circularity`: circularity given by $\frac{4\pi \cdot \text{area}}{\text{perimeter}^2}$;

- `solidity`: solidity given by the ratio $\frac{\text{area}}{\text{convex hull area}}$;
- `vertices`: approximate number of polygon vertices (`cv2.approxPolyDP` with $\epsilon = 0.02 \times$ perimeter).

5. **Sorting and Output:** sort objects in ascending order by the X position of the bounding rectangle (`bbox[0]`); in case of a tie, use the Y position (`bbox[1]`). Reassign `ids` from 1 to N and print the formatted table.
   - For sorting, use `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, with `medidas = mm.measure(img)`.

8.14.7.2 📌 Constraints and Sorting Rules

- **Object Sorting Rule:**

```
medidas.sort(key=lambda m: (m['bbox'][0], m['bbox'][1]))
```

- **Area Difference:** The area calculated by OpenCV (`cv2.contourArea`) measures the area of the continuous polygon delimited by the centers of border pixels, resulting in numeric values smaller than the simple discrete count of `1` pixels (`np.sum`).

8.14.7.3 🧠 Theoretical Foundation

Operation / Metric	Function in Filtering and Characterization
Morphological Opening ($\circ$)	Erosion followed by dilation: removes isolated bright noise (*salt*).
Morphological Closing ($\bullet$)	Dilation followed by erosion: fills small dark holes inside objects (*pepper*).
`cv2.boundingRect`	Returns (x, y, w, h), the smallest axis-aligned rectangle enclosing the object.
Circularity and Solidity	Describe the geometric compactness and convexity of the component.

8.14.7.4 📌 Examples

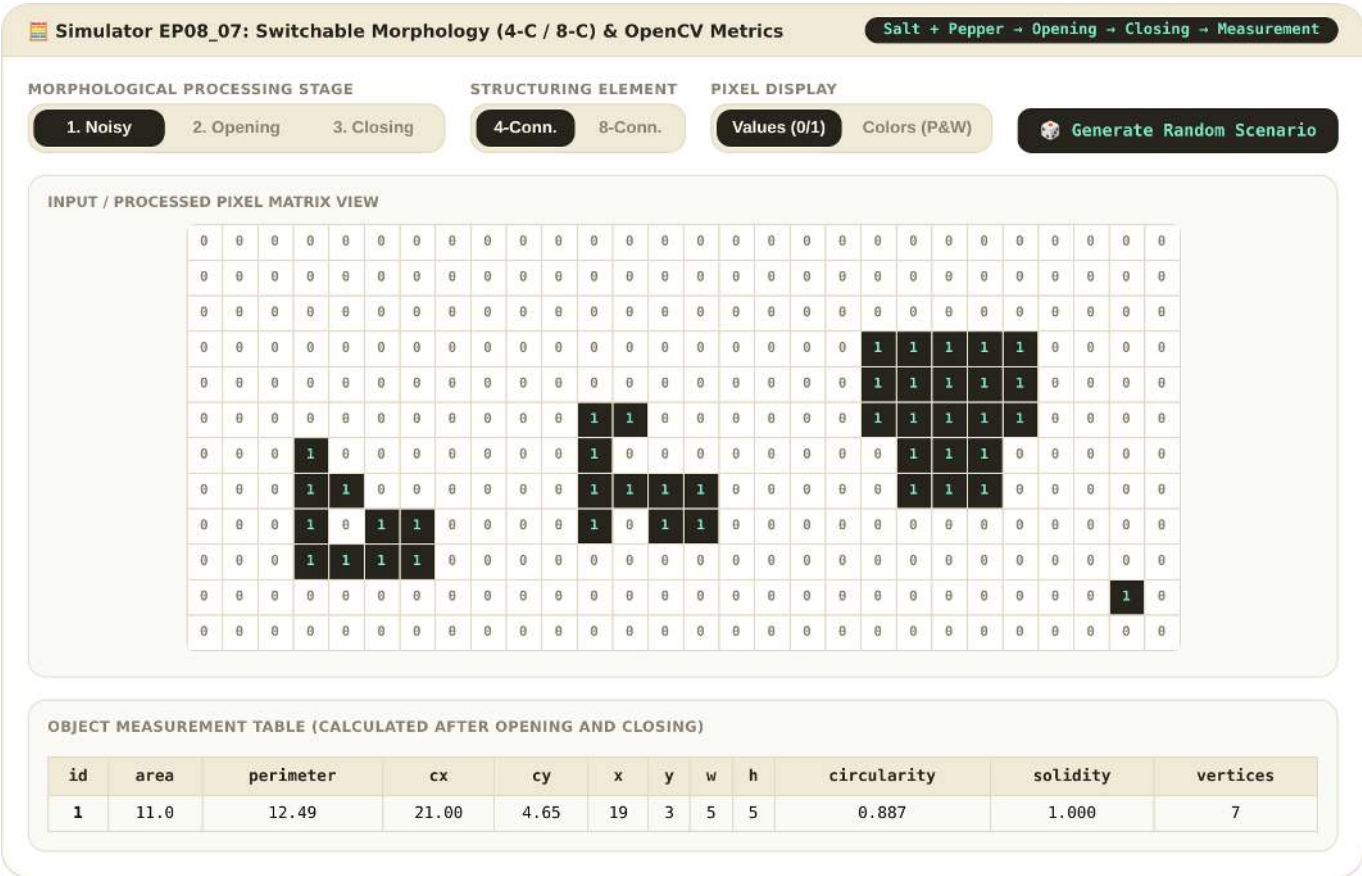
Input	Output
8 9 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0	id area perimeter cx cy x y w h circularity solidity vertices 1 9.0 12.0 3.5 2.0 3 1 4 3 0.79 1.000 4 2 4.0 8.0 7.5 5.5 7 5 2 2 0.79 1.000 4

```
%%writefile EP08_07.py
# Python code
```

```
Overwriting EP08_07.py
```

```
TestSuite("EP08_07.py").run()
```

```
<IPython.core.display.HTML object>
```



id	area	perimeter	cx	cy	x	y	w	h	circularity	solidity	vertices
1	11.0	12.49	21.00	4.65	19	3	5	5	0.887	1.000	7

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-ep0807.html

Figure 8.21: EP08_07 Simulator: Morphology with Configurable Connectivity and Measurement

8.14.8 EP08_08 🟡 Grayscale Image and Dynamic Thresholding

In this exercise, the input image is no longer strictly binary (`0/1`) but becomes a **grayscale image (8 bits,** $0 \ldots 255$**)**, where objects have an intermediate average intensity over a dark background (0), in addition to salt-and-pepper noise scattered throughout the entire image.

8.14.8.1 📋 Implementation Guidelines

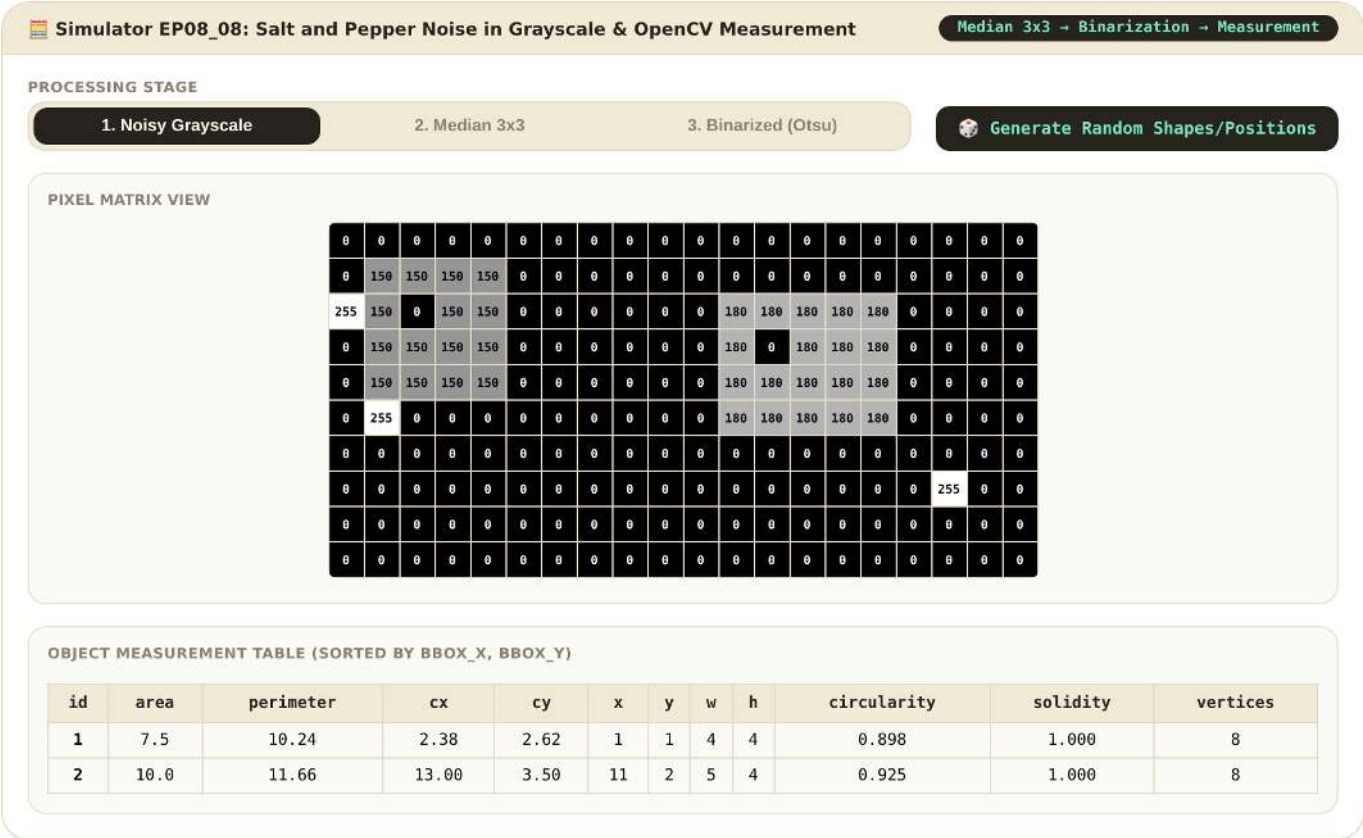
1. **Input:** read H and W on the first line, followed by the H lines with integer values from 0 to 255 in an $H \times W$ matrix.
2. **Preprocessing:**

- Apply a **Median filter** (3×3) to remove salt-and-pepper noise while keeping edges sharp.
- Apply **Otsu's thresholding** (or a fixed threshold $T = 60$) to binarize the clean image.

3. **Measurement and Output:** extract the contour of the objects, compute the geometric metrics (`area`, `perimeter`, `cx`, `cy`, `x`, `y`, `w`, `h`, `circularity`, `solidity`), and sort the objects by `bbox[0]` (with `bbox[1]` as a tiebreaker). Reassign `id` from 1 to N and print the table.
   - For sorting, use `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, with `medidas = mm.measure(img)`.

8.14.8.2 📌 Examples

Input	Output
16 32 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 0 0 0 0 0 0 0 … (binary image containing a square and a circle)	id area perimeter cx cy x y w h circularity solidity vertices 1 16.0 16.0 8.0 8.0 6 6 5 5 0.79 1.000 4 2 28.3 18.8 22.5 8.0 19 5 7 7 1.00 1.000 8



id	area	perimeter	cx	cy	x	y	w	h	circularity	solidity	vertices
1	7.5	10.24	2.38	2.62	1	1	4	4	0.898	1.000	8
2	10.0	11.66	13.00	3.50	11	2	5	4	0.925	1.000	8

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-ep0808.html

Figure 8.22: EP08_08 Simulator: Median Filtering in Grayscale and Object Measurement with OpenCV

```
%%writefile EP08_08.py
# Python code
```

```
Overwriting EP08_08.py
```

```
TestSuite("EP08_08.py").run()
```

```
<IPython.core.display.HTML object>
```

8.14.9 EP08_09 🟠 Illumination Gradient and Adaptive Thresholding

In this variation, the objects are immersed in a background with **non-uniform illumination (smooth illumination gradient)**. Simple single-value thresholding fails, requiring more robust preprocessing.

8.14.9.1 📋 Implementation Guidelines

1. **Input:** grayscale image $H \times W$ with background variation from 20 to 180.
2. **Preprocessing:**

- Apply **Adaptive Thresholding** (e.g., `cv2.adaptiveThreshold` with a Gaussian window of 15×15 and constant $C = 3$) to isolate the objects regardless of background variation.

  `cv2.adaptiveThreshold(img_gray, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, ksize, C) // 255`

  `ksize` and `C` are read after the image.

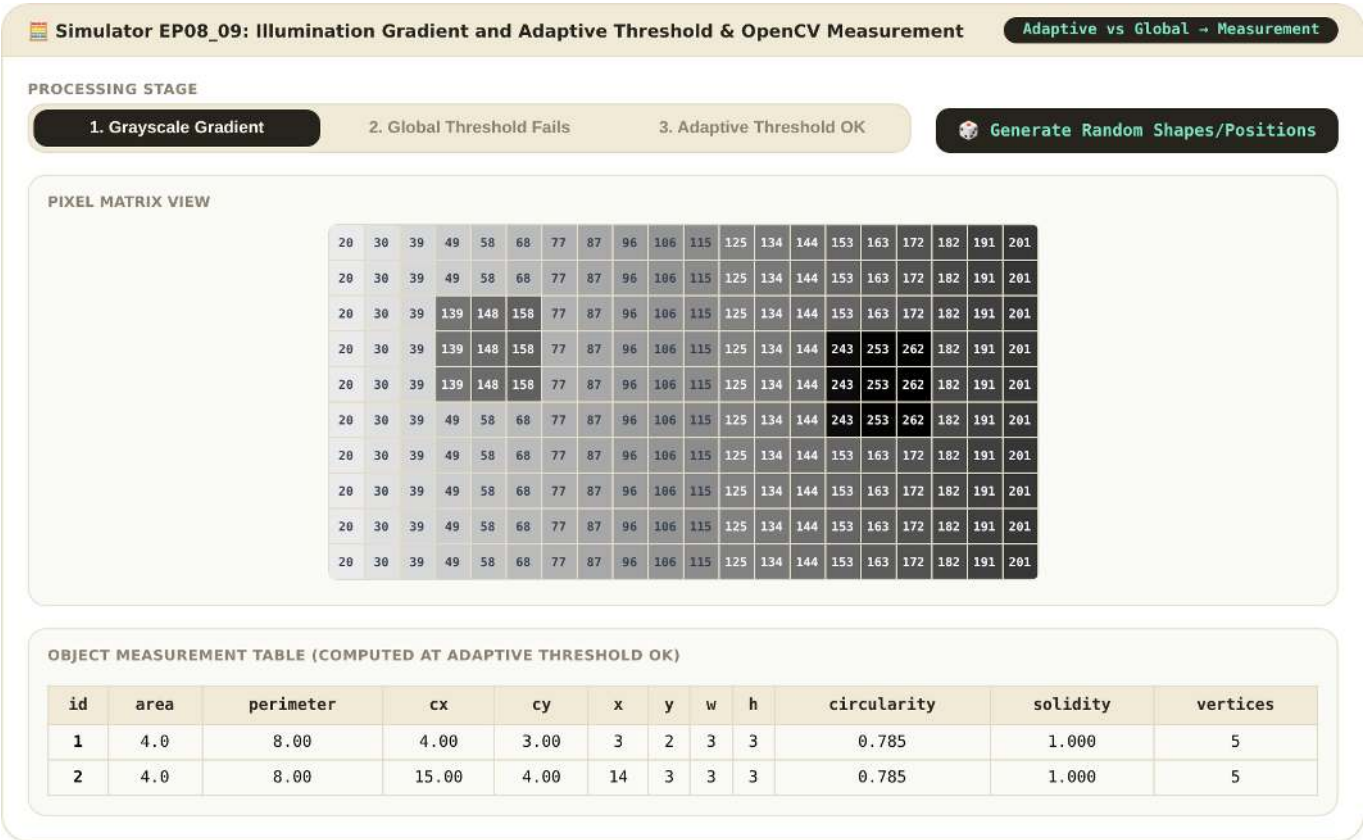
- Morphological **Closing** operation (3×3) to seal any gaps in the contours.

1. **Measurement and Classification:** extract the measurements.

2. **Sorting and Output:** sort by `(bbox[0], bbox[1])` and print the table including the `class` column.
   - To sort, use `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, with `medidas = mm.measure(img, precision=0.02)`.

8.14.9.2 📌 Examples

Input	Output
16 32 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 0 0 0 0 0 0 0 … k 20	id area perimeter cx cy x y w h circularity solidity vertices 1 9.0 12.0 10.0 5.0 8 3 5 5 0.79 1.000 4 2 28.3 18.8 25.0 12.0 22 9 7 7 1.00 1.000 3



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-ep0809.html

Figure 8.23: Simulator EP08_09: Illumination Gradient, Adaptive Threshold and OpenCV Measurement

```
%%writefile EP08_09.py
# Python code
```

```
Overwriting EP08_09.py
```

```
TestSuite("EP08_09.py").run()
```

```
<IPython.core.display.HTML object>
```

8.14.10 EP08_10 🔴 Low Contrast and Object Separation Topics (*Watershed* / Distance)

In this exercise, **some geometric objects are slightly touching (overlapping at the edges)**. Simply extracting contours would treat two objects as one.

8.14.10.1 📋 Implementation Guidelines

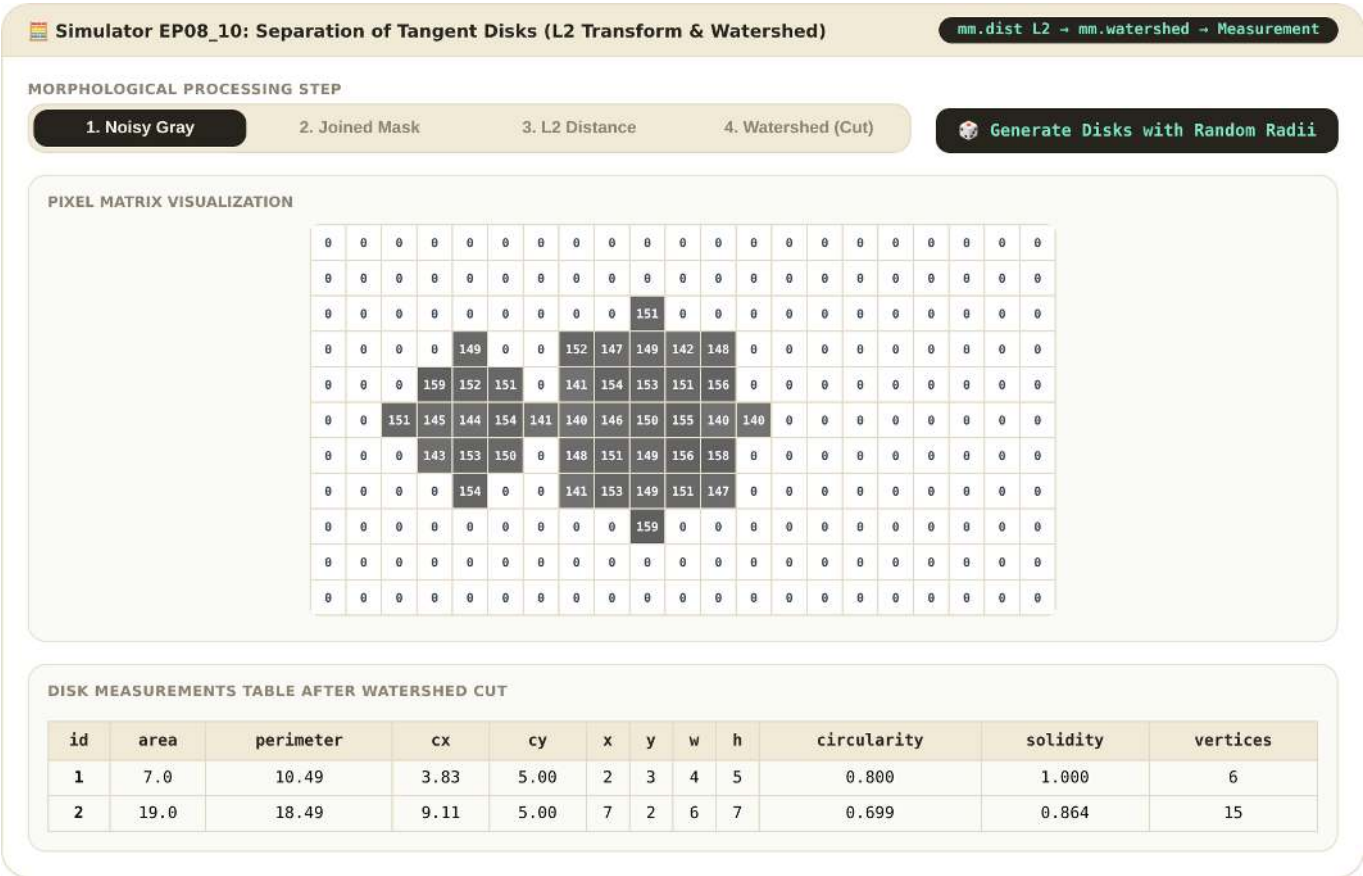
1. **Input:** an $H \times W$ grayscale matrix with objects of intensity $110 \ldots 140$ on a background of 0, with noise and a pair of tangent objects.
2. **Preprocessing and Separation:**

- Apply thresholding.
- Apply the **Distance Transform** (`mm.dist`).
- Obtain distance peaks to serve as markers in the **Watershed Transform** (`mm.watershed`), physically separating the touching objects in the mask. **Hint:** use `mm.regmax()` to obtain the local maxima and then label them with `mm.label0`.
- After the watershed, apply thresholding again with `mm.threshold(water,0)//255`.

3. **Connected Component Analysis:** measure each isolated region after the watershed.
4. **Output:** print the components sorted by `(bbox[0], bbox[1])` with their individual area, centroid, and solidity metrics.
   - For sorting, use `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, with `medidas = mm.measure(img, precision=0.02)`.

8.14.10.2 📌 Examples

Input	Output
16 16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 … (binary image containing two squares)	id area perimeter cx cy x y w h solidity 1 16.0 16.0 5.0 5.0 3 3 5 5 1.000 2 16.0 16.0 11.0 5.0 9 3 5 5 1.000



id	area	perimeter	cx	cy	x	y	w	h	circularity	solidity	vertices
1	7.0	10.49	3.83	5.00	2	3	4	5	0.800	1.000	6
2	19.0	18.49	9.11	5.00	7	2	6	7	0.699	0.864	15

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-ep0810.html

Figure 8.24: EP08_10 Simulator: Separation of Tangent Disks via L2 Distance Transform and *Watershed*

```
%%writefile EP08_10.py
# Python code
```

```
Overwriting EP08_10.py
```

```
TestSuite("EP08_10.py").run()
```

```
<IPython.core.display.HTML object>
```

8.14.11 EP08_11 🔴 Classification and Validation of Objects with Bounding Box Ground Truth

In this exercise, the goal is to process a grayscale image containing multiple geometric objects, extract their properties using `mm.measure`, and validate the detected bounding boxes against a real ground truth (GT) provided as input, using the IoU (Intersection over Union) metric.

8.14.11.1 📋 Implementation Guidelines

1. **Image Reading:** Read the dimensions $H \times W$ and the $H \times W$ pixel matrix of the grayscale image.
2. ***Morphological Pipeline:*** Binarize the image using the Otsu method (`mm.threshold`) and render the resulting binarized mask using `mm.drawImg`.
3. **Ground Truth Reading:**
   - Read the number G of ground truth bounding boxes.

- If $G > 0$, read G lines each containing 5 values: `id xmin_norm ymin_norm xmax_norm ymax_norm`.
- **Coordinate Conversion:** The ground truth coordinates are normalized in the range $[0.0, 1.0]$. To convert them to pixels on the image grid:

$$x_{\min} = \lfloor \text{xmin_norm} \times W \rfloor, \quad y_{\min} = \lfloor \text{ymin_norm} \times H \rfloor$$

$$w = \lfloor \text{xmax_norm} \times W \rfloor - x_{\min}, \quad h = \lfloor \text{ymax_norm} \times H \rfloor - y_{\min}$$

4. **Metric Extraction and IoU Calculation:**
   - Extract instance properties using `mm.measure(img_bin, precision=0.02)`.
   - For each detected bounding box (x, y, w, h), compute the IoU overlap with respect to the ground truth boxes and set `hits = 1` if there exists any match with IoU ≥ 0.50, otherwise set `hits = 0`.
5. **Output:** Sort the instances by position `(bbox[0], bbox[1])` and print the CSV table with the additional column `hits`.
   - For sorting, use `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, with `medidas = mm.measure(img)`.

8.14.11.2 🧠 Theoretical Foundation and Conversion

Concept	Formula / Operation	Description
Detected BBox	(x, y, w, h) via `mm.measure`	Bounding box computed on the discrete grid in integer pixels.
Ground Truth BBox (GT)	$(x_{\min}, y_{\min}, w, h)$ converted	Real box provided as input in relative coordinates $[0.0, 1.0]$.
IoU (Intersection over Union)	$\text{IoU} = \frac{\text{Area}(B_{\text{DET}} \cap B_{\text{GT}})}{\text{Area}(B_{\text{DET}} \cup B_{\text{GT}})}$	Evaluates the overlap rate of the boxes. It is considered valid if IoU ≥ 0.50.
Validation Status (`hits`)	1 if max(IoU) ≥ 0.50, otherwise 0	Binary indicator of detector correctness relative to the ground truth.

8.14.11.3 📦 Input and Output Specification (VPL)

Input:

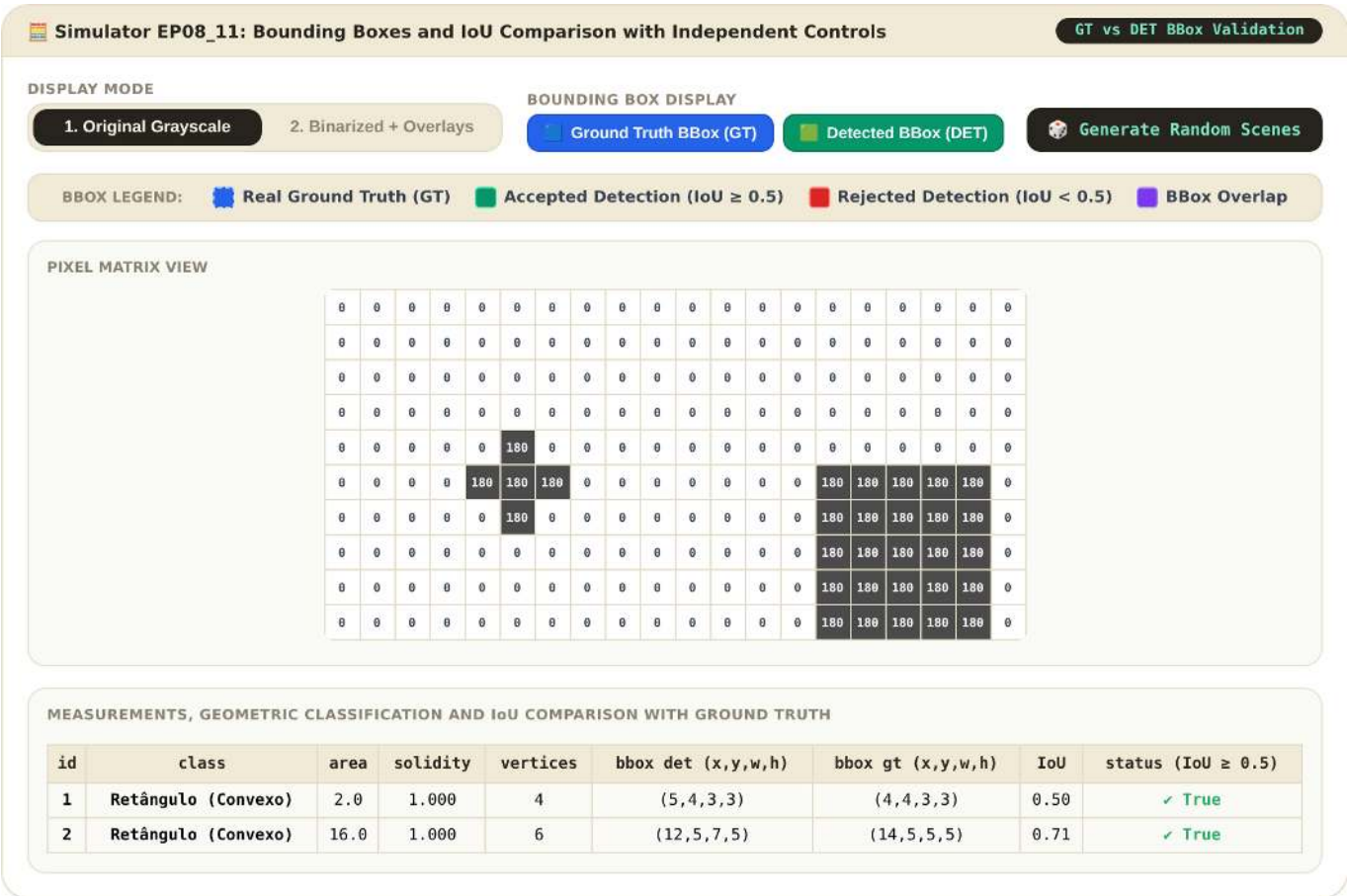
- **Line 1:** Integers H and W (dimensions of the matrix).
- **Next H lines:** W integers (0 to 255) representing the grayscale image.
- **Line $H + 2$:** Integer G (number of true ground truth boxes).
- **Next G lines:** 5 numeric values per line: `id xmin_norm ymin_norm xmax_norm ymax_norm` (where the coordinates are floating-point values between 0.0 and 1.0).

Output:

1. Rendered binarized matrix via `mm.drawImg(img_bin)`.
2. CSV header: `id,area,perimeter,cx,cy,x,y,w,h,circularity,solidity,vertices,hits`
3. One CSV line per detected object containing its formatted properties and the `hits` indicator (1 or 0).

8.14.11.4 📌 Examples

Input	Output
10 20	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 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 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 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
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0
0 0 0 180 0 0 0 0 0 0 0 0 180 180 180 0 0 0 0 0	0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0
0 0 180 180 180 0 0 0 0 0 0 0 180 180 180 0 0 0 0 0	0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0
0 0 0 180 0 0 0 0 0 0 0 0 180 180 180 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
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 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 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 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	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 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 0 0 0 0 0 0 0 0 0 0
2	id area perimeter cx cy x y w h circularity solidity
1 0.10 0.30 0.25 0.60	vertices hits
2 0.60 0.30 0.75 0.60	1 2.0 5.7 3.0 4.0 2 3 3 3 0.79 1.000 4 1
	2 4.0 8.0 13.0 4.0 12 3 3 3 0.79 1.000 4 1



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-ep0811.html

Figure 8.25: EP08_11 Simulator: Geometric Classification with Independent Controls for *BBox Overlays* (GT and DET)

```
%%writefile EP08_11.py
# Python code
```

```
Overwriting EP08_11.py
```

```
TestSuite("EP08_11.py").run()
```

```
<IPython.core.display.HTML object>
```

8.14.12 EP08_12 🔴 Instance Segmentation on Real Images: Geometric Objects

The classic segmentation example in this chapter separated coin "instances" by spatial disconnection in the binary mask resulting from Otsu's thresholding. In this exercise, you will apply the same idea—but now on a real image with varied geometric objects—by chaining together preprocessing, binarization, contour extraction (`cv2.findContours`), and validation of the result against a ground truth of bounding boxes.

Unlike the previous exercise (labeling on an already-prepared mask), here you start from the **original image**: the quality of your segmentation depends directly on the preprocessing choices (filtering, thresholding, morphological operations) made before labeling the components.

8.14.12.1 📋 Implementation Guidelines

1. **Input:** use the image `00000.jpg`.
2. **Preprocessing and segmentation:** apply the necessary steps (filtering, binarization, and morphological operations) to automatically separate the objects from the background, without manual cropping.
3. **Labeling and measurement:** for each segmented object, determine:
   - area;
   - center of mass (centroid);
   - type, according to the `obj2` set.
4. **Visual annotation:** write, inside each object, its area and the type abbreviation (`obj2`).
5. **Validation (IoU):** compute the Intersection over Union (IoU) between the detected bounding box (`cv2.boundingRect`) and the ground truth bounding box of the corresponding type. An object is considered correctly segmented only if there is exactly one bounding box of the correct type with **IoU 0.5**.
6. **Output:** print, for each detected object, its identifier, type, and whether it was successfully validated (`acertou=1`) or not. The output must follow the order of the `obj2` classes (0=Tria … 8=Cruz); within the same class, sort the objects by the vertical coordinate of the centroid (`cy`) in increasing order. Finally, print the overall accuracy.

8.14.12.2 📌 Computational Constraints

- **No manual cropping:** all segmentation must be performed on the full image.
- **Fixed set of classes:**

```
obj  = ['Triangulo','Quadrado','Pendagono','Hexagono','Heptagono','Circulo',
        'Elipse','Estrela','Cruz']
obj2 = ['Tria','Quad','Pent','Hexa','Hept','Circ','Elip','Estr','Cruz']
```

- **Image dimensions:** 608 × 608 pixels—used to denormalize the coordinates from the TXT file.
- **Center of mass validation:** an object is only considered correctly segmented if its centroid lies strictly inside the ground truth bounding box corresponding to the same object type.

8.14.12.3 🧠 Theoretical Background

Element	Role in instance segmentation
Preprocessing (filtering, thresholding)	Stage that produces the binary mask from the original intensity image
`cv2.findContours`	Extracts the contours of connected components in the binary mask

Element	Role in instance segmentation
Geometric moments (`cv2.moments`)	Allow computing the center of mass (centroid) of each contour
`approxPolyDP` / vertices	Aids in classifying the object type (approximate number of sides)
Bounding box validation	Confirms whether the segmented instance spatially corresponds to a ground truth object, measuring the method's accuracy

8.14.12.4 📌 Example of Expected Output

```
Objeto 1: tipo=Tria, validado=True
...
Acurácia: 88.89%
```

Fixed parameters for reproducibility: so that the output matches the automatic grading rubric, use exactly: minimum area filter of 300 pixels; `cv2.approxPolyDP` with `epsilon = 0.02 * perimeter`; solidity threshold of 0.92 and vertex count 9 (with 11 to distinguish Cruz from Estrela) for concave shapes; aspect ratio of 1.15 to distinguish Círculo from Elipse; IoU threshold of 0.5 for validation.

8.14.12.5 📌 Reference Files (`.jpg` and `.txt`)

For local debugging, two reference files are provided (included in this submission; when integrating them into the chapter repository, save them in `all/cap08/dados/EP08/`):

- 📥 **Image (`00000.jpg`)**: image of geometric objects used as the exercise input. The goal is to automatically segment each object, determine its type, and compute its measurements.
- 📥 **Ground truth (`00000.txt`)**: file containing the normalized bounding boxes of the objects in the image. Each line contains the class identifier and the normalized coordinates of the upper-left and lower-right corners, used to automatically validate the segmentation.

Figure 8.26 shows the input image and the same image with the bounding boxes drawn from the ground truth file.

```python
import os
import urllib.request
from morph import mm

def garantir_e_baixar(nome):
    pasta = "dados/EP12"
    caminho = os.path.join(pasta, nome)

    os.makedirs(pasta, exist_ok=True)

    if not os.path.exists(caminho):
        url = (
            "https://raw.githubusercontent.com/"
            "fzampirolli/pdi-vc/master/all/cap08/dados/EP12/"
            + nome
        )
        print(f"Downloading {nome}...")
        urllib.request.urlretrieve(url, caminho)

    return caminho

img_arq = garantir_e_baixar("00000.jpg")
txt_arq = garantir_e_baixar("00000.txt")
```

```
img = mm.read(img_arq)
img_bb = mm.showBoundBox(img, txt_arq, fmt="yolo", show=False)

mm.show(
    [img, img_bb],
    titles=[
        "Original image",
        "Answer key bounding boxes"
    ],
    cols=2,
    figsize=(10,5)
)
```

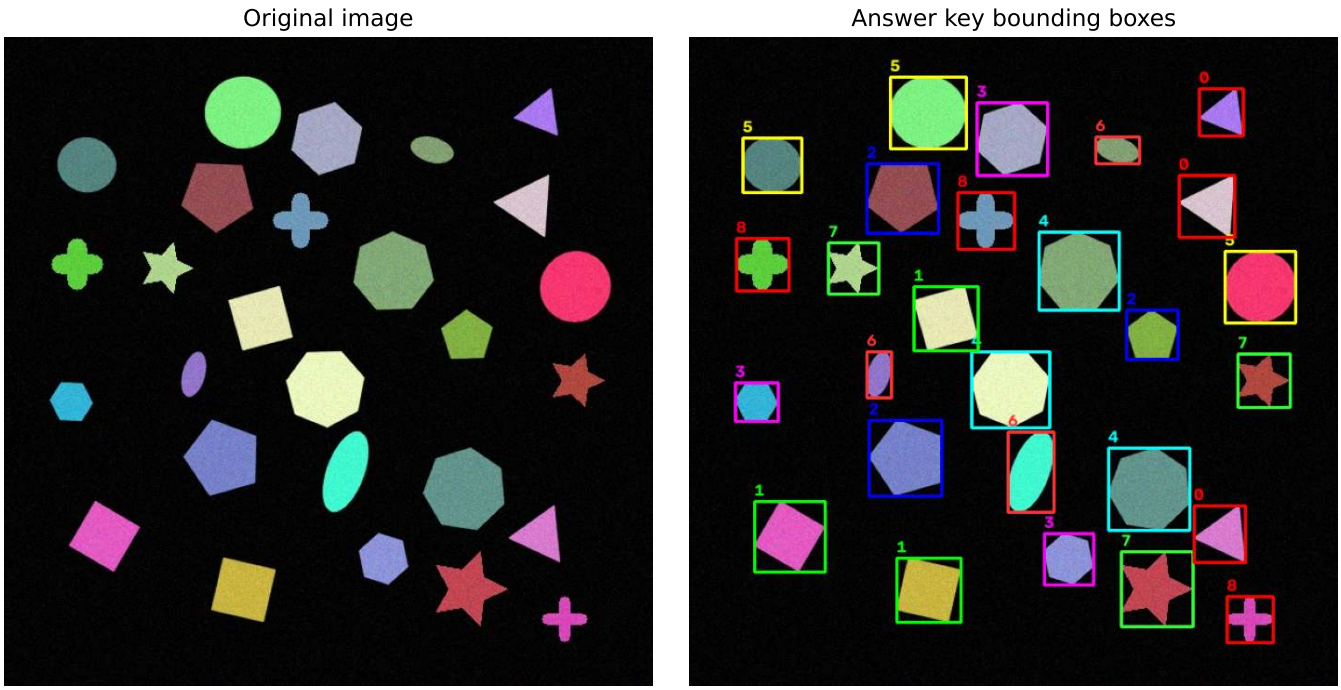


Figure 8.26: Simulator EP08_12: Image used in EP08_12. On the left, the original image. On the right, the image with the *bounding boxes* from the answer key file.

```
%%writefile EP08_12.py
# Python code
```

```
Overwriting EP08_12.py
```

```
TestSuite("EP08_12.py").run()
```

```
<IPython.core.display.HTML object>
```

Simulator EP08_12: Segmentation Accuracy on Multiple Objects

hit if IoU ≥ threshold AND correct type

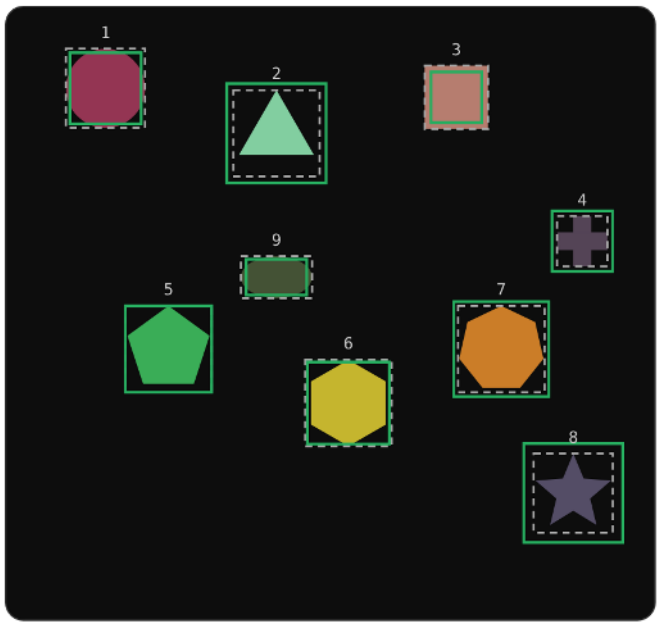
Each shape has a *bounding box* ground truth (dashed rectangle, tight around the shape) and a *bounding box* detected (solid rectangle, offset/noisy). Adjust the noise, bias, and IoU threshold to see the validation change.

segmentation noise (px, max jitter per side) 0

systematic bias in x (px) 0

IoU threshold 0.50

simulate classification error (2 objects with swapped types)



id	true type	detected type	IoU	≥ threshold	hit
1	Circ	Circ	0.81	sim	✔
2	Tria	Tria	0.76	sim	✔
3	Quad	Quad	0.64	sim	✔
4	Cruz	Cruz	0.69	sim	✔
5	Pent	Pent	1.00	sim	✔
6	Hexa	Hexa	0.90	sim	✔
7	Hept	Hept	0.83	sim	✔
8	Estr	Estr	0.64	sim	✔
9	Elip	Elip	0.72	sim	✔

Objetos validados: 9 / 9 → Acurácia = 100.0%

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap08/sim-ep0812.html

Figure 8.27: EP08_12 Simulator: Segmentation Accuracy in Multiple Objects (IoU)

Chapter 9

Deep Learning for Computer Vision

Run Colab Open GitHub

The previous chapters established the foundations of Computer Vision (CV) through classical methods of feature extraction and representation. In **Chapter 7**, descriptors such as *Local Binary Patterns* (*LBP*) and *Histogram of Oriented Gradients* (*HOG*) showed how textures and shapes can be encoded by manually designed descriptors. In **Chapter 8**, algorithms such as *Oriented FAST and Rotated BRIEF* (*ORB*) and the *Haar Cascade* detector extended this principle to tasks of matching, detection, and object recognition.

These techniques remain relevant for their interpretability and computational efficiency, but they rely on a prior step of manually defining descriptors, known as **feature engineering**. This dependence limits the model's adaptation to scenarios for which the descriptor was not designed.

Deep Learning proposes an alternative: instead of manually specifying the relevant features, the model automatically learns representations from the data during training—a process known as **representation learning** (GOODFELLOW, 2016). In CV, this strategy is implemented mainly through **Convolutional Neural Networks** (CNNs), in which convolutional filters no longer have fixed coefficients and are instead adjusted by optimization algorithms.

Although they represent a shift in the construction of pattern recognition systems, CNNs preserve concepts already studied in this book: the convolution, presented in **Chapter 3**, remains the operation responsible for local feature extraction, now applied with learned coefficients rather than designed ones.

It is worth noting that the goal of this chapter is not to exhaustively explore the theory of Deep Learning, but rather to offer an overview of its foundations and to demonstrate how these architectures are applied in the context of CV. Readers interested in a deeper theoretical and conceptual understanding of the field should refer to specialized references in the literature, such as Goodfellow (2016) and Lecun (2015).

9.1 Chapter Objectives

By the end of this chapter, the student should be able to:

- Relate the convolution learned by CNNs to the convolution of fixed *kernels* presented in **Chapter 3**;
- Describe the basic architecture of a CNN and the function of its main layers;
- Implement, train, and evaluate CNN models for image classification;
- Apply **transfer learning** to adapt pre-trained models to new problems;
- Use pre-trained models in classification, object detection, and segmentation tasks;
- Implement, train, and evaluate a *U-Net* architecture for semantic segmentation, comparing it to classical approaches;
- Prepare annotated datasets and integrate them into a training *pipeline* through platforms such as **Roboflow**;
- Integrate computational geometry and Deep Learning in augmented reality and photogrammetry applications.

Figure 9.1 synthesizes the organization of the concepts studied in this chapter and the relationships among them.

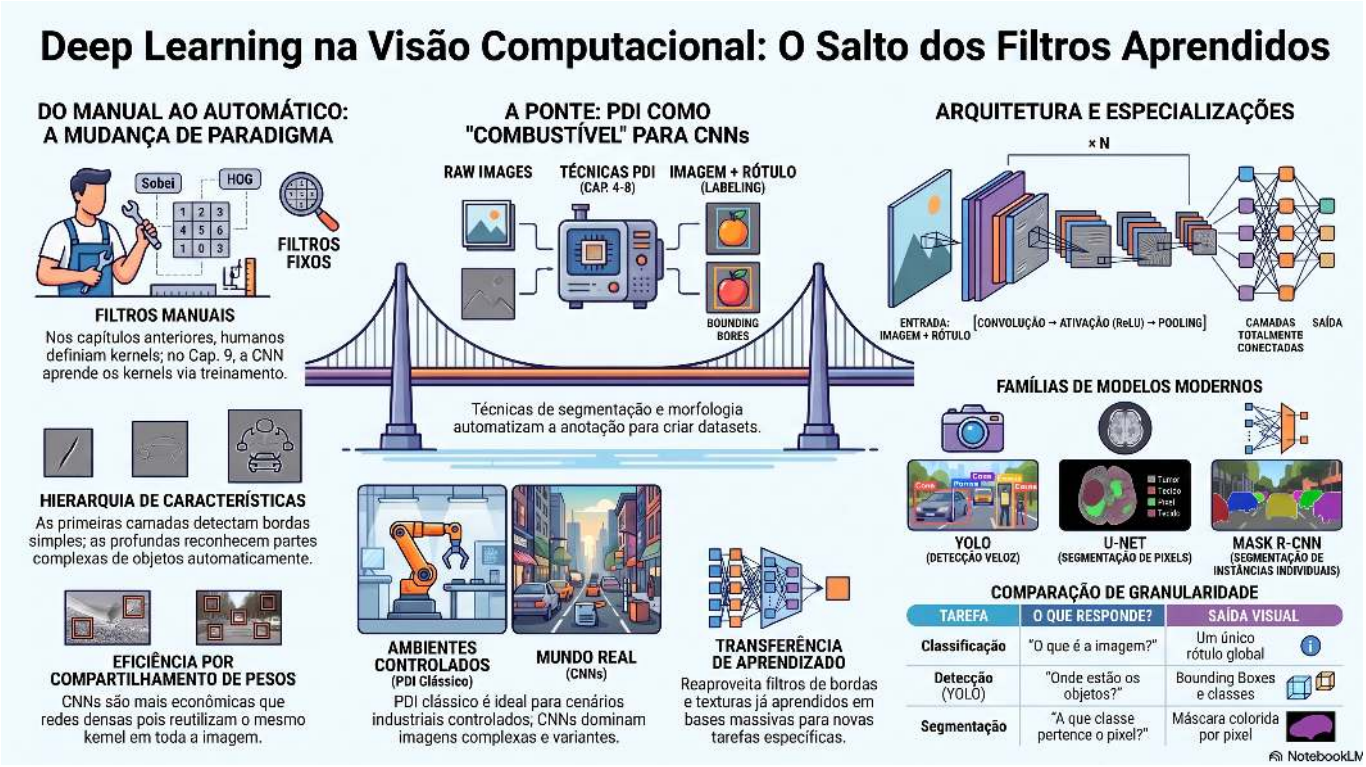


Figure 9.1: Overview of the main concepts addressed in this chapter. **Source:** prepared with the aid of *Gemini Notebook* ({GOOGLE}, 2025).

9.2 Overview: Classification, Detection, and Segmentation

Computer vision tasks primarily differ in the information produced as output. **Classification** assigns a single label to the entire image; **object detection** locates and identifies objects present in the scene; **segmentation** associates a class with each pixel and, in some approaches, distinguishes different instances of the same category.

Table 9.1 summarizes the tasks studied throughout the book, indicating the question each one answers and the granularity of the information produced.

Table 9.1: Comparison among the main computer vision tasks according to the granularity of the information produced.

Task	Question Answered	Output Granularity	Chapter
Classification	"What is the class of this image?"	A single label for the entire image	7 and 9
Object detection	"Which objects exist and where are they?"	Class and bounding box for each object	8 and 9
Semantic segmentation	"To which class does each pixel belong?"	A class label for each pixel	8 and 9
Instance segmentation	"Which pixels belong to each object?"	One label per pixel for each instance	8 and 9
Panoptic segmentation	"What is the class and identity of each object?"	Class and instance identifier for each pixel	8 and 9

These tasks represent increasing levels of image interpretation: classification describes the scene globally, detection adds the location of objects, and segmentation produces a detailed spatial representation, allowing

each region to be analyzed individually. This chapter focuses first on classification by CNNs, then extends the same principles to detection and segmentation.

9.3 Environment Setup

The examples in this chapter use **PyTorch**, a *framework* widely employed in the development and training of Deep Learning models. The following code checks the availability of the required libraries and automatically installs those not yet present in the execution environment.

If **PyTorch** is not installed, a version compatible with the available hardware is automatically selected: the version with **CUDA** support, if an NVIDIA GPU is available, or the CPU execution version, otherwise.

Next, the environment is initialized with the import of the libraries used throughout the chapter, the definition of a random seed to favor experiment reproducibility, and the retrieval of the `morph.py` file — the didactic morphological processing library already used in previous chapters —, if it is not yet available in the working directory.

```
import contextlib, importlib, importlib.metadata, importlib.util
import io, os, random, shutil, subprocess, sys, urllib.request, warnings

# Suppresses PyTorch warnings and general warnings
warnings.filterwarnings("ignore", category=UserWarning)

url = ("https://raw.githubusercontent.com/fzampirolli/"
       "pdi-vc/master/morph/config.py")
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config

# Silences stdout and stderr both at Python level and at OS File Descriptor level
def setup_silencioso():
    with open(os.devnull, "w") as fnull:
        old_out = os.dup(1)
        old_err = os.dup(2)
        try:
            os.dup2(fnull.fileno(), 1)
            os.dup2(fnull.fileno(), 2)
            with contextlib.redirect_stdout(fnull), contextlib.redirect_stderr(fnull):
                config.setup()
        finally:
            os.dup2(old_out, 1)
            os.dup2(old_err, 2)
            os.close(old_out)
            os.close(old_err)

setup_silencioso()
from morph import mm


def setup_cap09():
    """Installs missing libs for this chapter in a 100% silent way."""
    pkgs = {
        "skimage": "scikit-image", "numpy": "numpy",
        "sklearn": "scikit-learn", "matplotlib": "matplotlib",
        "torchviz": "torchviz", "ultralytics": "ultralytics",
        "roboflow": "roboflow",
    }
    for mod, pkg in pkgs.items():
        if importlib.util.find_spec(mod) is None:
```

```
            subprocess.run([sys.executable, "-m", "pip", "install", "-q", pkg],
                           stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)

    if importlib.util.find_spec("torch") is None:
        args = (["torch", "torchvision"] if shutil.which("nvidia-smi")
                else ["--index-url",
                      "https://download.pytorch.org/whl/cpu",
                      "torch", "torchvision"])
        subprocess.run([sys.executable, "-m", "pip", "install", "-q", *args],
                       stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)

    if not shutil.which("dot") and shutil.which("apt-get"):
        subprocess.run(["apt-get", "install", "-y", "-qq", "graphviz"],
                       stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL, check=False)


setup_cap09()

import cv2, numpy as np, torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import torchvision.models as models
import torchvision.transforms as T
from PIL import Image
from skimage import data as skdata
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, TensorDataset
from torchvision.datasets import OxfordIIITPet
from torchvision.models.detection import (
    fasterrcnn_resnet50_fpn, FasterRCNN_ResNet50_FPN_Weights)
from torchvision.models.segmentation import deeplabv3_resnet50
from torchvision.transforms.functional import to_tensor
from torchviz import make_dot
from ultralytics import YOLO

torch.manual_seed(42)
FLAG_LIMPAR_DADOS = False
device = "cuda" if torch.cuda.is_available() else "cpu"
gpu = f" ({torch.cuda.get_device_name(0)})" if device == "cuda" else ""
ver = importlib.metadata.version("ultralytics")
print(f"  Environment ready. OpenCV {cv2.__version__} | "
      f"morph {getattr(mm, '__version__', 'local_file')} | "
      f"PyTorch {torch.__version__} | Ultralytics {ver} | {device}{gpu}")
```

```
  Environment ready. OpenCV 5.0.0 | morph local_file | PyTorch 2.6.0+cu124 | Ultralytics
8.4.113 | cuda (NVIDIA GeForce GTX TITAN X)
```

9.4 Fundamentals of Deep Learning for CV

CNNs are the primary Deep Learning architecture applied to image analysis. Their operation is based on the composition of convolutional operations organized into successive layers, in which the filters learned during training transform the image into progressively more abstract representations. This section presents the fundamental concepts that connect the spatial convolution studied previously to modern CV models, including hierarchical feature extraction, the training process, and the use of pre-trained models.

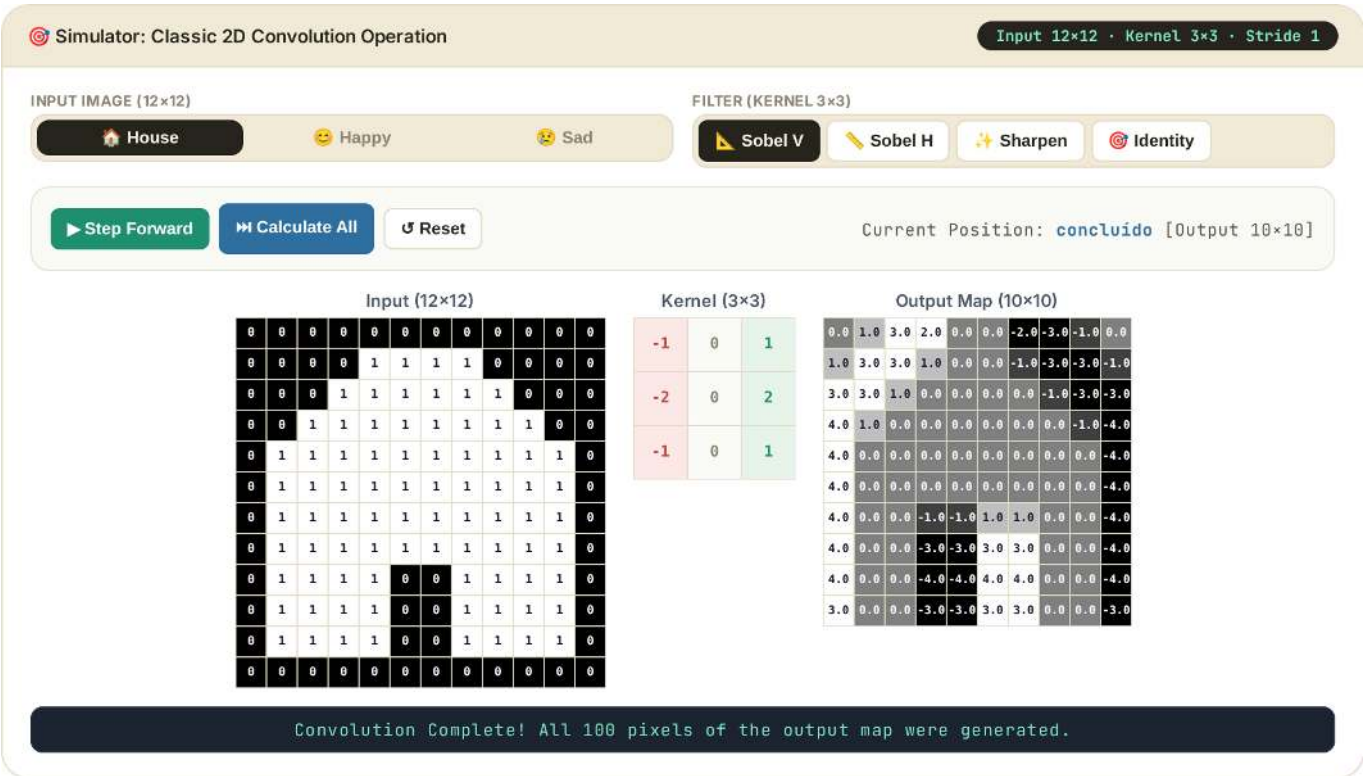
9.4.1 From Fixed Convolution to Learned Convolution

Chapter 3 presented spatial convolution with fixed *kernels*, such as the Sobel operators, designed to highlight specific features of an image. In **Chapters 7** and **8**, the same principle underpinned descriptors such as *HOG*, *LBP*, and *ORB*, as well as the *Haar Cascade* detector: in all these cases, the filters are defined before the execution of the algorithm and remain unchanged during processing.

Figure 9.2 revisits how spatial convolution works: the simulator allows selecting different *kernels* and following the displacement of the convolution window over an image. At each position, the *kernel* coefficients combine with the local neighborhood of the image — known as the **receptive field** — to produce a value in the **feature map**, also illustrating weight sharing.

CNNs preserve this operation but replace fixed *kernels* with **learned filters**: instead of predefined coefficients, the network adjusts these values during training from labeled examples, seeking to minimize a **loss function**, which measures the difference between the model's predictions and the expected responses.

The essential difference between classical methods and CNNs, therefore, lies not in the convolution operation itself, but in how the filters are obtained: whereas the former use manually designed filters, CNNs learn, from the training data, representations suited to the task.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-09-convolucao.html

Figure 9.2: Interactive simulator of classic 2D convolution: choose among the three synthetic 12×12 input images (house, happy or sad face) and a 3×3 filter (Sobel V, Sobel H, Sharpen, or Identity) and step forward to observe how the local inner products of the receptive field build the feature map cell by cell.

To understand how this learning occurs, it is necessary to study the basic processing unit of neural networks: the **artificial neuron**.

9.4.2 Artificial Neuron

The **artificial neuron** is the fundamental processing unit of a neural network. Its first mathematical model—a set of combined inputs compared against a threshold—was proposed by Mcculloch (1943), still without any learning mechanism. The **Perceptron** (ROSENBLATT, 1958) advanced this formulation by introducing a rule for adjusting weights based on examples, becoming the first artificial neuron model capable

of learning and the foundation of modern **Deep Learning** architectures. The term Deep Learning refers to the use of networks with multiple processing layers, capable of learning hierarchical representations of data: the first layers learn simple features, such as edges and textures, and deeper layers progressively combine these representations to identify more complex structures and objects.

Each neuron receives a set of inputs, computes a linear combination of these values, and applies an **activation function**, producing a single output value. Mathematically, the linear combination is given by

$$z = \sum_{i=1}^{n} w_i x_i + b,$$

where x_i represent the inputs, w_i the weights associated with each input, and b the **bias**. The neuron's output is obtained by applying the activation function:

$$y = f(z).$$

In CNNs, this principle assumes different forms depending on the layer. In **convolutional layers**, each neuron processes only a small region of the input, called the **receptive field**, preserving the spatial organization of the image. In **fully connected layers**, each neuron receives all outputs from the previous layer, combining the extracted features to produce the network's final output, such as the class assigned to the image.

Figure 9.3 illustrates the functioning of an artificial neuron: the simulator allows modifying the inputs (x_1 and x_2), the weights (w_1 and w_2), the bias (b), and the activation function, observing in real time the calculation of the linear combination and the corresponding output.

In a CNN, thousands of neurons are organized into layers with specific functions: the first ones are responsible for feature extraction through convolution, and the last ones perform classification based on the learned features.

9.4.3 Convolutional Layer

The **convolutional layer** is responsible for extracting features from the image. Each filter generates a **feature map**, whose intensity at each position indicates the filter's response to the corresponding region of the input.

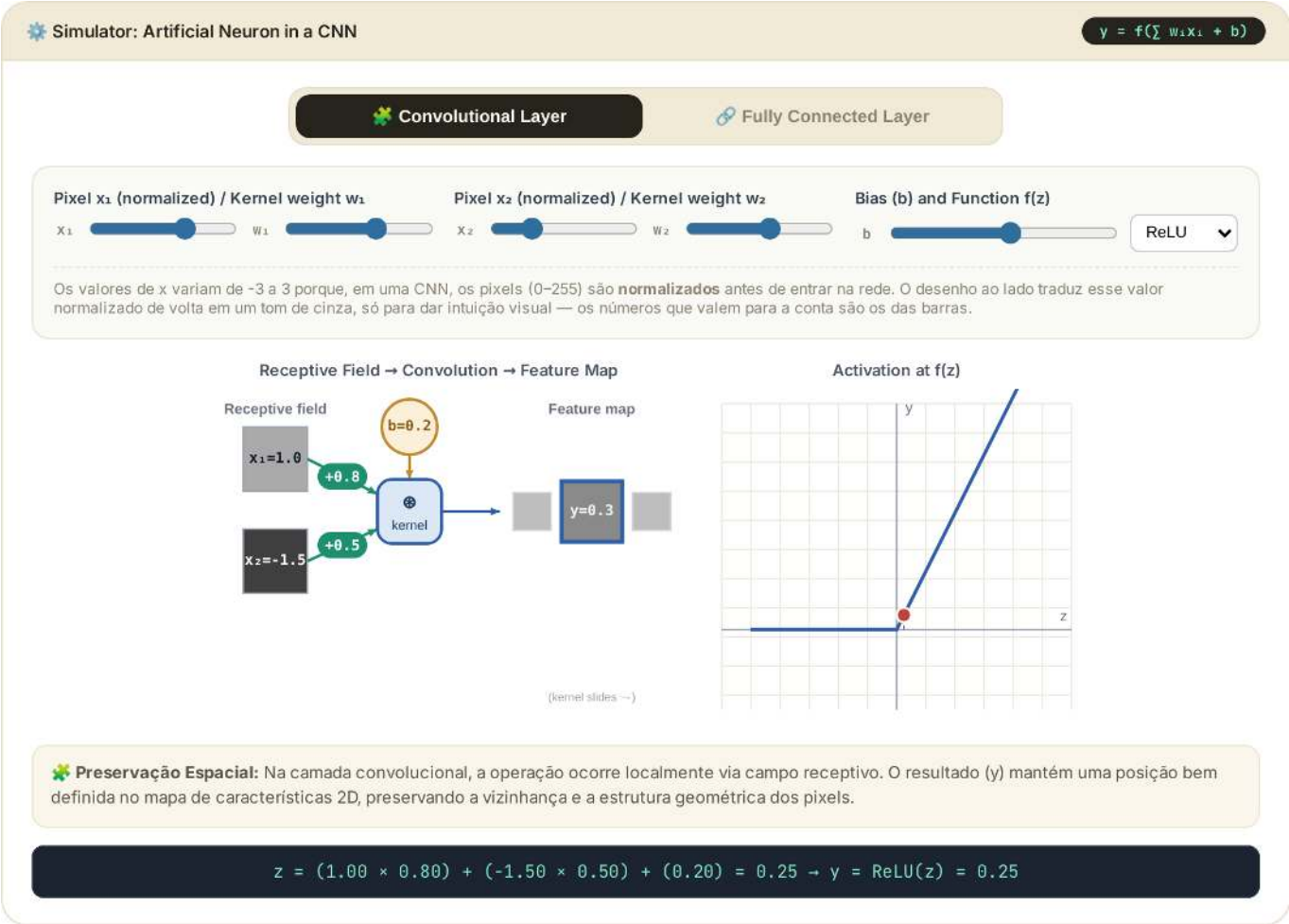
The operation performed follows the same principle of shifting and local combination presented in **Chapter 3** for spatial convolution. Considering a kernel K of dimension $k \times k$, the value produced at position (i, j) is given by

$$F(i, j) = \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} K(u, v)\, I(i + u, j + v).$$

It is worth noting a terminological distinction: the expression above formally corresponds to a **cross-correlation**, rather than strict mathematical convolution, which requires reflecting the kernel before combination. Most Deep Learning frameworks, including **PyTorch**, implement this operation without reflection and designate it, by convention, as convolution — a convention also adopted in this chapter. This difference has no practical effect on training, since the kernel coefficients are learned rather than imposed beforehand.

The main difference from classical methods therefore lies in obtaining the kernel K: in traditional filters, its coefficients are manually defined to highlight specific image features; in CNNs, the coefficients are initialized automatically and adjusted during training through **error backpropagation**, making each filter specialized in identifying patterns relevant to the task under study.

Two concepts characterize this layer:



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-09-neuronio.html

Figure 9.3: Interactive simulator of the Artificial Neuron in a CNN context: toggle between a convolutional layer neuron (where x_i are pixel intensities in a receptive field and w_i are kernel weights) and a fully connected layer neuron, adjusting inputs, weights, bias, and activation function to visualize the computation of z and output y in real time.

- **Weight sharing**: the same filter is applied at all positions of the image, significantly reducing the number of model parameters.
- **Receptive field**: each convolutional neuron processes only a small neighborhood of the image, preserving the spatial structure of the data.

By stacking multiple convolutional layers, the network learns a **feature hierarchy**: the early layers tend to detect simple patterns, such as edges and textures, and the deeper layers combine this information to represent progressively more complex structures. After convolution, the feature map is subjected to an activation function, introducing nonlinearity into the model and expanding its capacity to represent complex relationships among input variables.

Figure 9.4 presents this layer interactively.

9.4.4 Activation Function

Convolution is a linear operation. To enable the network to model nonlinear relationships between inputs and outputs, an **activation function** is applied after each convolutional layer.

The most commonly used function in CNNs is **ReLU** (*Rectified Linear Unit*), defined as

$$\mathrm{ReLU}(x) = \max(0, x).$$

This function preserves positive values and replaces negative values with zero, introducing nonlinearity into the model and facilitating the training of deep networks at low computational cost.

Figure 9.5 illustrates the behavior of **ReLU** applied to both individual values and a feature map, allowing a comparison of the output before and after activation.

The feature maps resulting from convolution and activation preserve the spatial structure of the image. In many architectures, the next step reduces their resolution through a *pooling* operation.

9.4.5 *Pooling*

The ***pooling*** layer reduces the spatial resolution of feature maps while preserving the most relevant information for subsequent processing stages. The most commonly used operation is ***max-pooling***, which selects the largest value within each window of the image:

$$P(i, j) = \max_{(u,v)\in \mathrm{window}(i,j)} F(u, v).$$

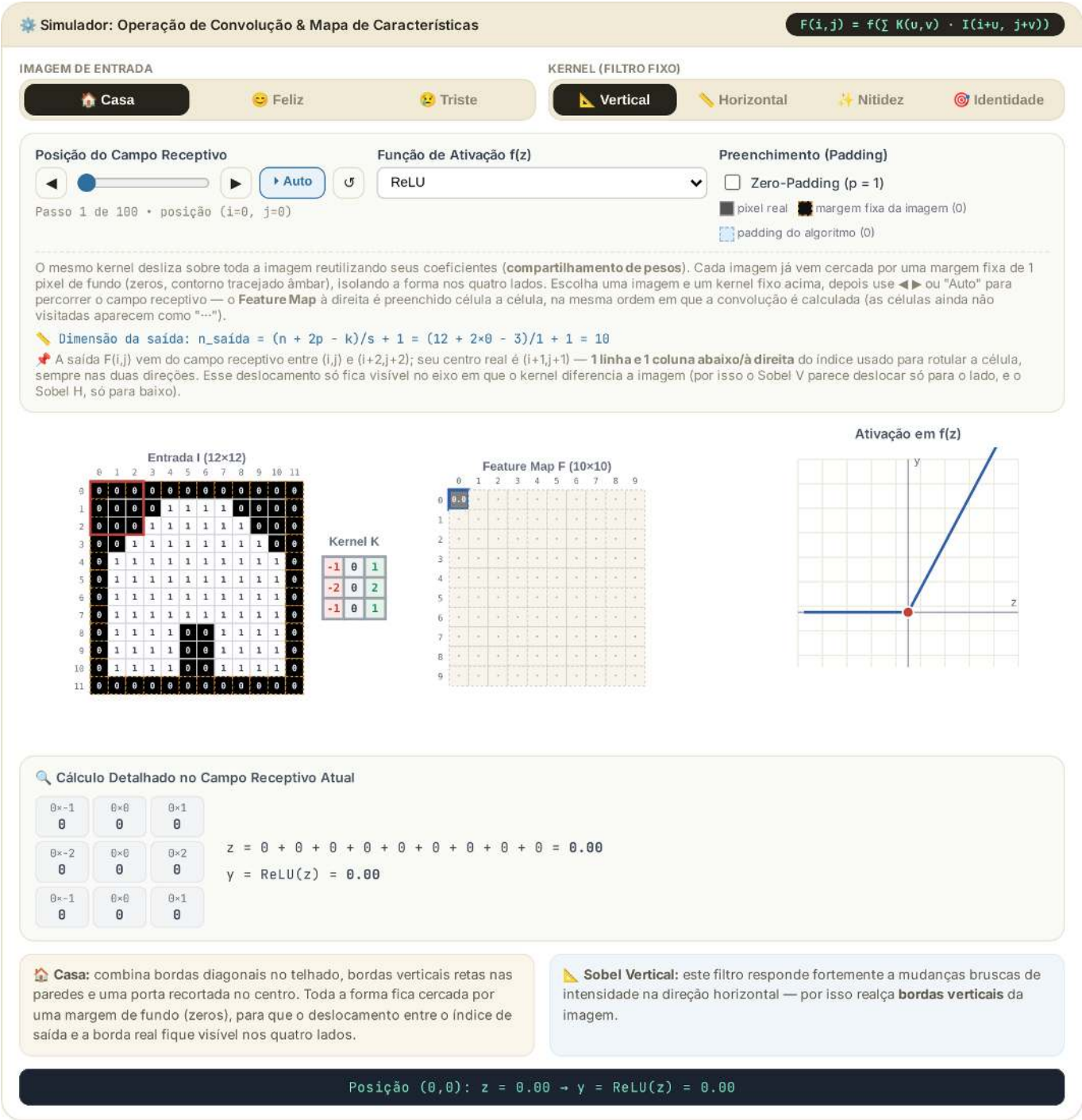
This reduction lowers the computational cost of subsequent layers and makes the representation more robust to small variations in the position of patterns present in the image.

Figure 9.6 illustrates this operation on an 8×8 pixel feature map, reduced to 4×4 using 2×2 windows with a stride of 2, alternating between **max-pooling** and **average-pooling** — which, instead of the maximum, computes the average of the values within the corresponding window.

Together, convolution, activation function, and *pooling* form the basic building block used in constructing a CNN.

9.4.6 Training Neural Networks: How CNNs Learn

A CNN learns by automatically adjusting its parameters — the coefficients of the convolutional filters, the weights of the fully connected layers, and the biases (*biases*) — from labeled examples. This training is iterative and involves three steps: measuring the error produced by the network through a **loss function**, calculating how this error depends on each parameter through **backpropagation**, and updating the parameters with an **optimization algorithm** (*optimizer*).



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-09-camada-conv.html

Figure 9.4: Interactive Convolutional Layer Simulator: choose among three 12×12 input images (house, happy face, or sad face) to observe how the same fixed *kernels* react to different edges and shapes. Navigate the receptive field with the buttons or automatic playback, adjust the activation function and toggle zero-padding, while the feature map is revealed cell by cell, with detailed term-by-term calculation and the output dimension formula in real time.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-09-relu.html

Figure 9.5: Interactive simulator of the ReLU activation function: drag the control to see how negative values are zeroed out and positive values are preserved, both in the curve and in a real feature map.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-09-pooling.html

Figure 9.6: Interactive *pooling* simulator: choose between *max-pooling* and *average-pooling* and step forward to observe the reduction of the spatial resolution of the feature map.

9.4.6.1 Loss Function

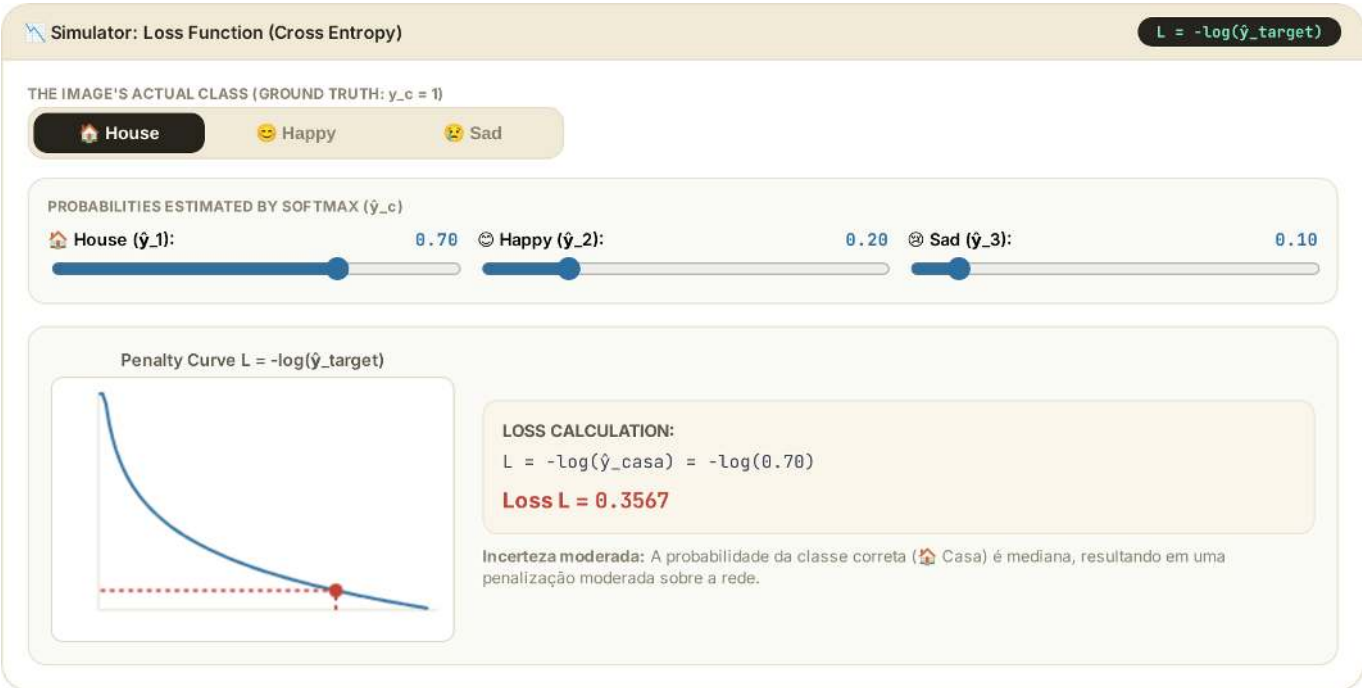
The **loss function** quantifies the difference between the network's prediction and the correct answer, referred to as the **ground truth**. The result is a scalar L: the lower the loss, the closer the prediction is to the expected answer.

In multiclass classification problems, the most commonly used function is **Cross-Entropy Loss**, applied to the probabilities produced by the **Softmax** layer:

$$L = -\sum_{c=1}^{C} y_c \log(\hat{y}_c),$$

where C is the number of classes, y_c is the actual label in *one-hot* encoding, and $\hat{y}_c$ is the predicted probability for class c. The loss approaches zero when the network assigns high probability to the correct class and grows rapidly as that probability decreases.

Figure 9.7 illustrates this behavior: the simulator allows selecting the correct class and altering the probabilities produced by *Softmax*, showing in real time the variation of the loss function.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-09-loss.html

Figure 9.7: Interactive simulator of the Loss Function (*Cross-Entropy*): select the real class of the image (House, Happy or Sad) and adjust the probabilities estimated by Softmax to visualize the calculation of the scalar penalty and the graph of the negative logarithm in real time.

9.4.6.2 Backpropagation

After computing the loss, it is necessary to determine how each network parameter contributes to this result. This step is carried out by **backpropagation**, which applies the **Chain Rule** of differential calculus to obtain the gradient of the loss function with respect to each parameter.

For a parameter w, this gradient is given by

$$\frac{\partial L}{\partial w}.$$

The gradient indicates how the loss varies with respect to small changes in w: a positive gradient indicates that increasing w increases the loss, and a negative gradient indicates the opposite effect.

Figure 9.8 presents this process visually, showing the propagation of the gradient from the output layer to the earliest convolutional layers.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-09-backprop.html

Figure 9.8: Interactive *Backpropagation* simulator: advance through the Chain Rule steps to follow the error signal flow in the reverse direction of the network, observing the computation of partial derivatives of the gradient at each layer.

9.4.6.3 Optimization Algorithms

After computing the gradients, an **optimization algorithm** (*optimizer*) updates the network parameters to reduce the loss function. In deep networks, this search occurs in a high-dimensional space and, in general, a **non-convex** one, which makes optimization a challenging problem.

To facilitate understanding, Figure 9.9 uses a **simplified loss surface**, with a global minimum, a local minimum, and a barrier between these regions. The **global minimum** corresponds to the smallest value of the loss function and represents the best set of network parameters; a **local minimum** also exhibits low loss but may be far from the best solution. When optimization becomes trapped in a local minimum, the adjustments to filters, weights, and biases become very small, and training stops before reaching a model with lower error.

Stochastic Gradient Descent (*SGD*)

Stochastic Gradient Descent (*SGD*) updates the parameters in the opposite direction of the gradient:

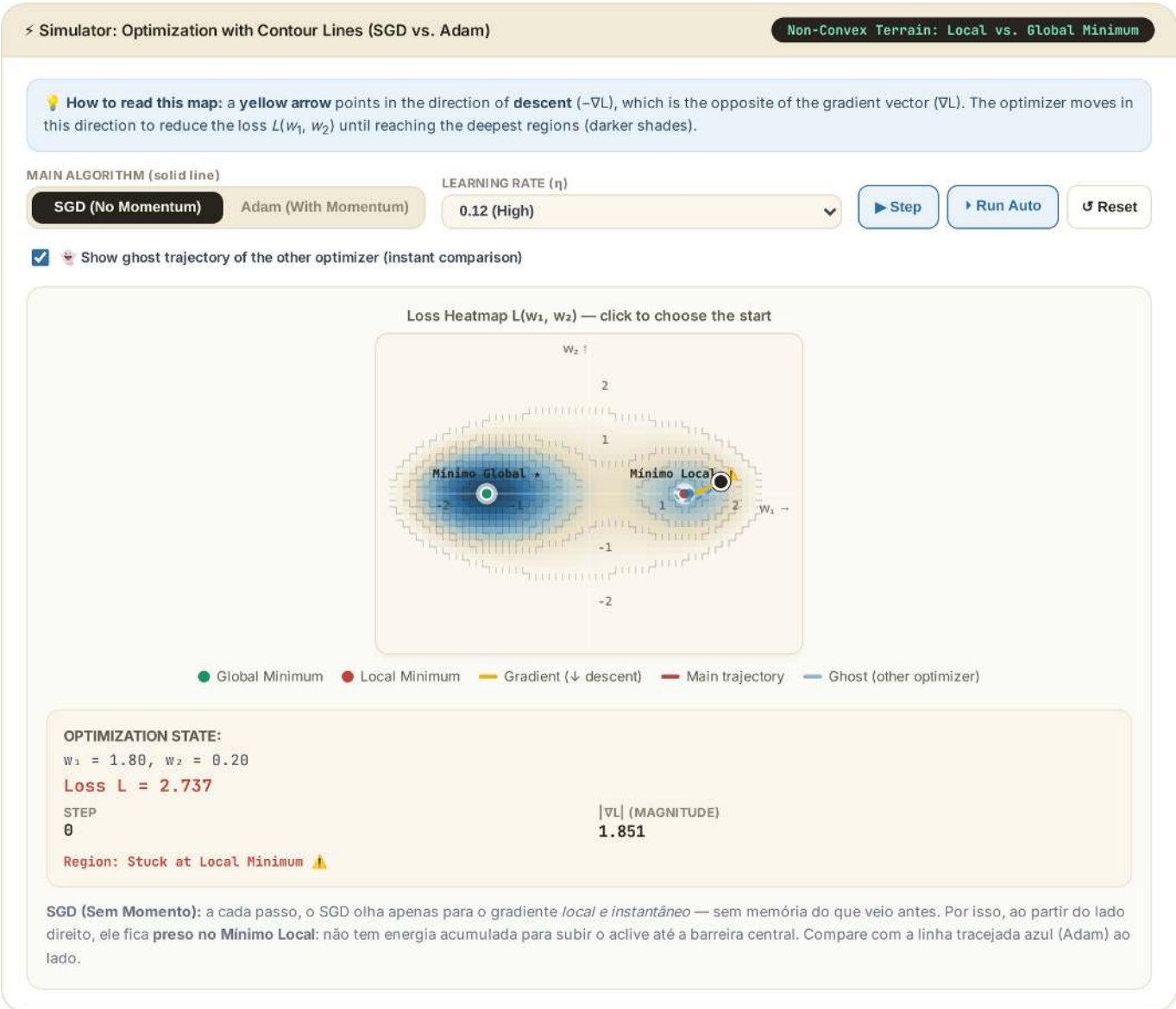
$$w_{\text{new}} = w_{\text{current}} - \eta \frac{\partial L}{\partial w},$$

where η is the **learning rate**, responsible for controlling the size of the update. *SGD* uses only the gradient of the current iteration; when the search reaches a local minimum, the gradients become very small and updates practically cease.

Adaptive Optimizers: *Adam*

Adam (*Adaptive Moment Estimation*) combines adaptive estimates of the first and second moments of the gradients (KINGMA, 2015), adjusting the learning rate for each parameter individually. In many cases, this adaptation helps overcome local minima that would trap *SGD*.

Figure 9.9 compares the trajectory of *SGD* and *Adam* over the same non-convex loss surface.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-09-opt.html

Figure 9.9: Interactive simulator of Optimization Algorithms: compare the trajectory of SGD and Adam over a non-convex loss surface with a heatmap and contour lines. The solid line shows the selected optimizer advancing step by step; the dashed line shows, for instant comparison, the full path that the other optimizer would take from the same starting point. Notice how SGD gets stuck in the Local Minimum on the right, while Adam may or may not cross the central barrier depending on accumulated momentum and learning rate. Click anywhere on the plot to reset the starting point of the weights.

9.4.7 Architecture of a CNN

A CNN for image classification combines the layers presented in the previous sections. During the **forward pass**, the image successively traverses the convolutional layers, the activation functions, the *pooling* operations, the **Flatten** step, the fully connected layers, and finally the **Softmax** layer, which produces the class probabilities. During training, this prediction is compared to the correct label to compute the loss function, perform backpropagation, and update the parameters through an optimization algorithm.

Figure 9.10 presents this processing and training flow.

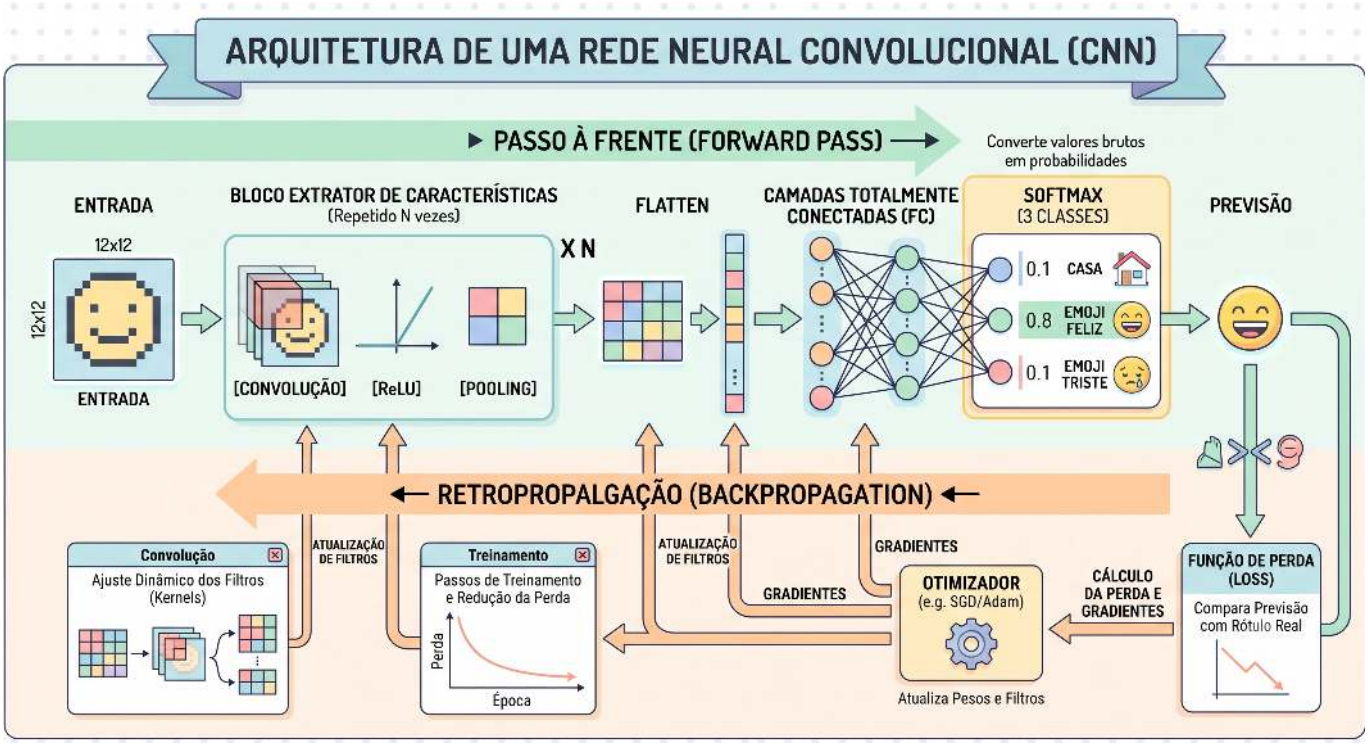


Figure 9.10: Simplified architecture of a CNN for image classification, highlighting the *forward pass* and the training steps through the loss function, backpropagation, and the optimization algorithm.

After the last convolutional block, the **Flatten** operation reorganizes the feature maps into a one-dimensional vector, which feeds the **fully connected layers**, responsible for combining the extracted features to produce the scores (*logits*) for each class. The **Softmax** layer converts these scores into a probability distribution, used both for classification and for computing the loss function during training.

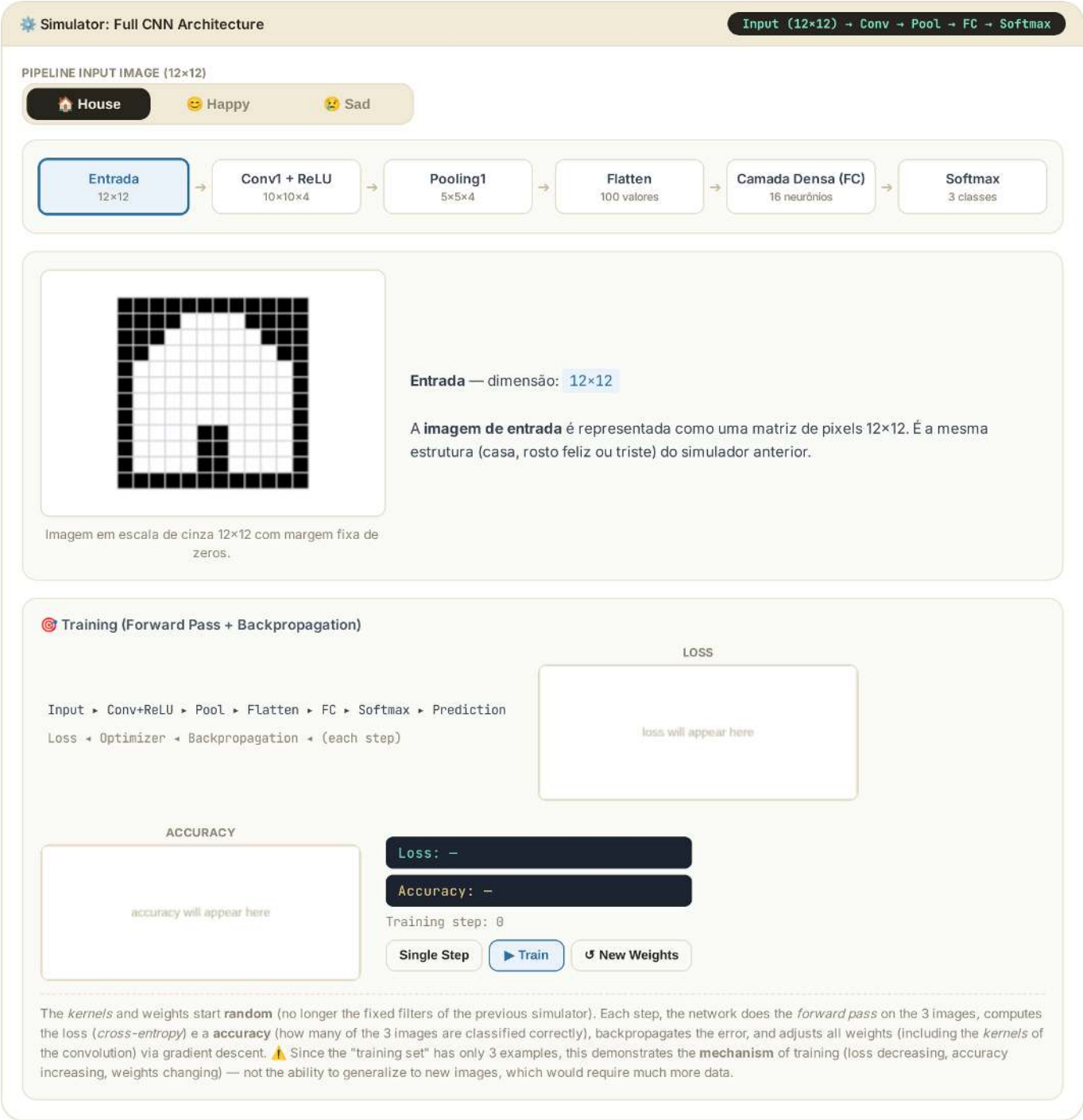
Figure 9.11 presents an interactive version of this architecture, allowing successive training steps to be executed and enabling observation of the loss reduction, the backpropagation of gradients, and the update of the network filters.

9.4.8 How the Gradient Adjusts Convolution Kernels

Understanding the learning process in a Convolutional Neural Network (CNN) requires elucidating a fundamental mechanism: **how the random coefficients of an initial filter transform into precise detectors of edges, textures, and complex patterns.**

The answer lies in the principle of **weight sharing**. During the forward pass, the same filter of dimension 3×3 slides across the entire extent of the input image. Consequently, each kernel weight—such as the element $K[0][0]$ in the upper-left corner—is reused multiple times across the different spatial regions of the input data.

During backpropagation, this reuse establishes a direct dynamic: **each spatial position processed by the filter generates an individual contribution ("vote") for the update of the respective weight.**



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-09-arquitetura.html

Figure 9.11: Interactive simulator of a CNN architecture: choose one of the 12×12 input images (house, happy face, or sad face), click on each block of the *pipeline* — Input, *Conv+ReLU*, *Pooling*, *Flatten*, *FC*, and *Softmax* — and run real training steps (*forward pass* + backpropagation) to observe the loss and accuracy evolving, the *kernels* being adjusted, and the *Softmax* starting to point to the correct class.

9.4.8.1 The Intuition Behind the Calculation

Let $Z[r][c]$ be the pre-activation feature map at position (r, c) of the sliding window, obtained by the cross-correlation between the kernel K and the input X:

$$Z[r][c] = \sum_{k_r} \sum_{k_c} K[k_r][k_c] \cdot X[r + k_r][c + k_c]$$

Applying the chain rule to determine the contribution of a specific weight $K[k_r][k_c]$ to the loss function L, the following steps are obtained:

1. **Local Error (dZ):** At each position (r, c), the partial derivative of the loss function with respect to the pre-activation is computed:
$$dZ[r][c] = \frac{\partial L}{\partial Z[r][c]}$$
which quantifies the responsibility of that specific position for the total network error (L).
2. **Weight Contribution:** Since $\frac{\partial Z[r][c]}{\partial K[k_r][k_c]} = X[r + k_r][c + k_c]$, the influence of a specific weight $K[k_r][k_c]$ on the error at position (r, c) is obtained by multiplying the local error $dZ[r][c]$ by the value of the input pixel aligned with that weight at the time of the computation:
$$dZ[r][c] \cdot X[r + k_r][c + k_c]$$
3. **Gradient Accumulation:** The final gradient of the weight corresponds to the **sum of the contributions ("votes") from all positions** traversed by the sliding window:
$$\frac{\partial L}{\partial K[k_r][k_c]} = \sum_{(r,c)} dZ[r][c] \cdot X[r + k_r][c + k_c]$$

This formulation ensures direct parity between the analytical derivation and the values computed in the gradient inspection simulator (Figure 9.12).

9.4.8.2 The Role of the ReLU Function as a "Relevance Filter"

The application of the **ReLU** activation function ($\max(0, z)$) immediately after the convolution introduces a selectivity property to the gradient:

- **Positive Activation ($Z[r][c] > 0$):** The derivative of ReLU is 1. The local error is propagated in full ($dZ \neq 0$), allowing the position to contribute to the kernel weight updates.
- **Inactive Activation ($Z[r][c] \leq 0$):** The derivative of ReLU is 0. The local error is nullified ($dZ = 0$), suppressing the position's contribution to the final gradient.

  Pedagogical note: ReLU ensures that only the spatial regions that produced active responses during the forward pass have the ability to modify the kernel weights in the backpropagation process.

9.4.8.3 Weight Update via Gradient Descent

After consolidating the accumulated gradients from all positions, the weight update occurs according to the Stochastic Gradient Descent (SGD) algorithm:

$$K[k_r][k_c] \leftarrow K[k_r][k_c] - \eta \cdot \frac{\partial L}{\partial K[k_r][k_c]}$$

where η denotes the **learning rate**.

- If the sum of the gradients is **positive**, the weight value is reduced.
- If the sum is **negative**, the weight value is increased.

9.4.8.4 Exploring the Interactive Simulator

🔗 From Global Architecture to Gradient Inspection

In the architecture simulator (Figure 9.11), one observes the error dZ derived from the complete multilayer backpropagation, originating from the softmax cross-entropy loss over the 12×12 input images.
To enable the analytical verification of the gradient without the overhead of 100 convolution positions and multilayer backpropagation, the kernel learning simulator (Figure 9.12) adopts a reduced inspection model (6×6). In this scenario, the problem is simplified by replacing the complex classification with a **scalar calibration target**: the filter is adjusted to produce a predefined accumulated response (target $= 9$) when identifying a specific pattern (such as a 45-degree edge). The gradient accumulation mechanism ($dZ \cdot X$) remains strictly identical in both formulations.

To inspect this dynamic at the numerical level, use the simulator in Figure 9.12:

- **Input Image (X):** 6×6 matrix.
- **Convolutional Filter (K):** 3×3 matrix (9 weights).
- **Output Map (Z / A):** 4×4 matrix (16 window positions).
- **Loss Function (L):** Defined by $L = \frac{1}{2}(S - \text{target})^2$, where $S = \sum A[r][c]$ represents the global sum of post-ReLU activations.

  The role of target $= 9$: The scalar value target $= 9$ represents the ideal "activation energy" stipulated for the image with a diagonal edge. Since the map A has 16 positions, this value is equivalent to seeking an average response of $\frac{9}{16} \approx 0.56$ per activated pixel. When $S > 9$, the network identifies that the filter is reacting with excessive intensity to the pattern, generating an error $dZ > 0$ that forces a reduction in the weights K. When $S < 9$, the weights are increased to amplify the signal.

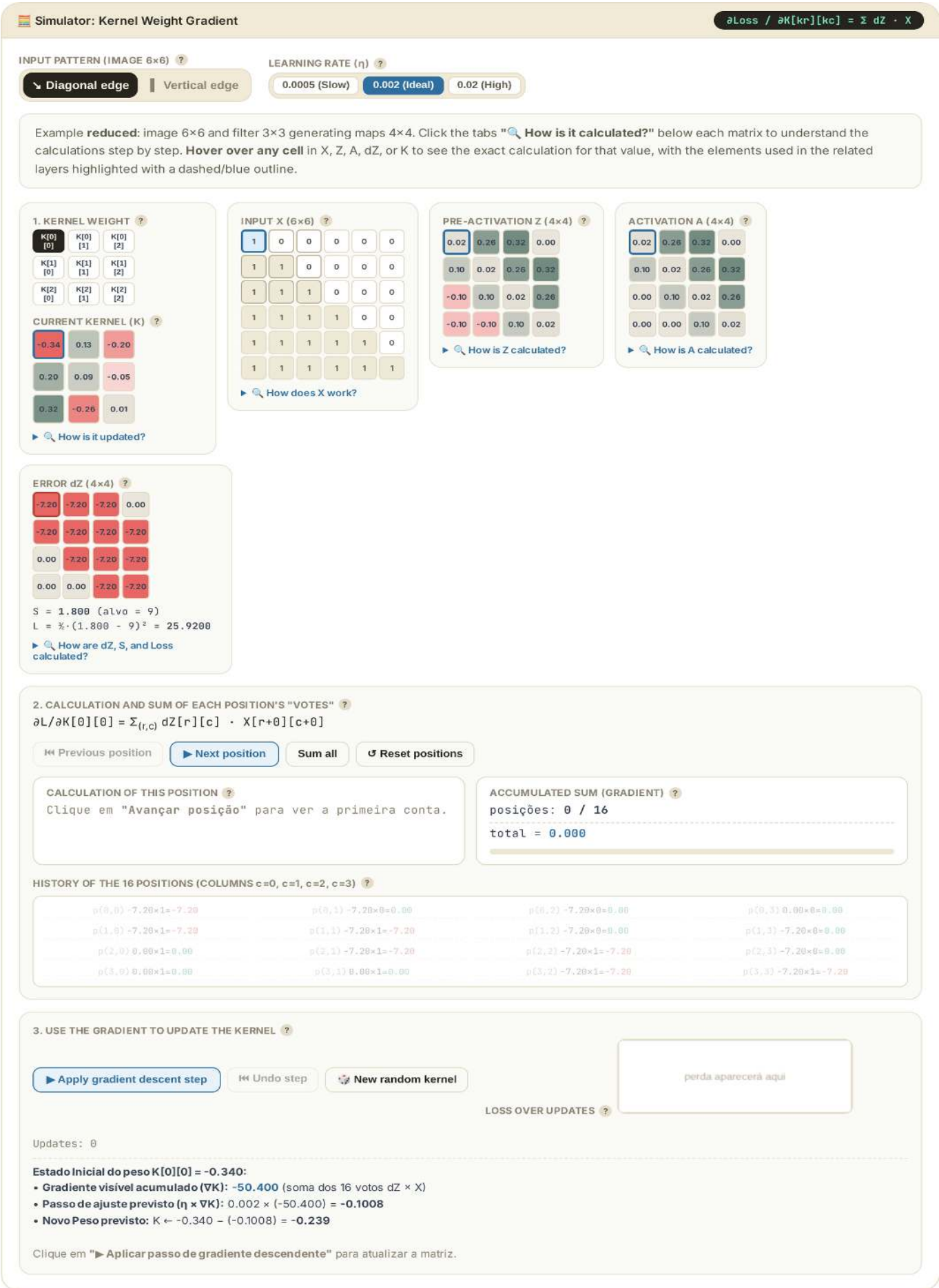
Suggested Experimentation Script:

1. **Weight Selection:** In the 3×3 grid, choose the weight to be analyzed (e.g., $K[0][0]$).
2. **Window Scan:** Use the **"▷ Advance position"** button to track the window's displacement across the 16 spatial positions. Note the visual highlight on the input map cell that aligns the pixel X with the selected weight.
3. **Local Vote Analysis:** Examine the product of the local error and the input pixel ($dZ \cdot X$) in the position calculation panel.
4. **History Verification:** Track the consolidation of the 16 partial results organized in the four history columns, observing the accumulation of the final gradient.
5. **Kernel Update:** Click on **"▷ Apply gradient descent step"** to visualize the convergence of the loss curve and the adaptation of the random kernel to the selected input pattern.

🧠 Synthesis — From convolution to representation learning

The simulators in this section demonstrate, in a sequential manner, how a CNN transforms an input image into a probabilistic estimate and how its parameters are optimized during training:

- **Convolution:** applies filters to the image to extract local features, generating feature maps through **weight sharing**.
- **ReLU:** introduces nonlinearity into the system, enabling the modeling of complex relationships among data.
- ***Pooling*:** reduces the spatial resolution of the feature maps, decreasing computational cost and conferring invariance to small local translations.
- ***Flatten*:** reorganizes the multidimensional maps into a one-dimensional vector for feeding subsequent layers.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-09-gradiente-kernel.html

Figure 9.12: Interactive simulator for computing the gradient of a weight from the convolutional *kernel*.

- **Fully connected layer:** combines the extracted features to produce raw scores (*logits*) associated with each class.
- **Softmax:** converts the *logits* into a normalized probability distribution.
- **Loss function:** compares the predicted distribution with the ground truth, quantifying the network's error as a scalar.
- **Backpropagation:** applies the chain rule to compute the partial derivative (gradient) of the loss function with respect to each trainable parameter.
- **Optimizer:** updates the coefficients of filters, weights, and biases in the direction opposite to the gradient, reducing the loss at each iteration.

Throughout the iterations, the convolutional filters evolve from stochastic values into specialized detectors: early layers learn low-level visual primitives (such as edges and textures), while deeper layers consolidate these representations into abstract and semantic structures.

9.5 Practical Applications in CV

After the theoretical consolidation of the fundamentals of CNNs and the visual verification of each of their elementary operations through interactive simulators, it becomes essential to observe the integration of these steps into complete programming *pipelines*.

In the following sections, the theory is translated into executable code in **PyTorch**, exploring the three fundamental tasks of CV: **classification**, **object detection**, and **semantic segmentation**. This practical progression allows for analysis ranging from the construction of a convolutional architecture trained from scratch to the application of advanced **transfer learning** strategies in pre-trained models for synthetic and real datasets.

9.5.1 Image Classification with CNNs

Image classification is one of the most traditional applications of CNNs. In this task, the goal is to assign a single label to the input image, such as identifying an object category, an animal species, or a diagnostic class. To achieve this, the CNN progressively transforms pixel values into higher-level abstract representations, combining convolutional layers, activation functions, and spatial reduction operations until it produces a probability distribution across the possible classes. In this section, we present the basic architecture of a classifying CNN, the data transformation flow along the network, and the training process for adjusting the learned parameters.

9.5.1.1 Training a CNN from Scratch on Digits

To establish a direct comparison with the approaches presented in Chapter 7, this section develops a CNN trained on the same handwritten digits dataset (`load_digits`). The fundamental difference lies in the representation step: while classical methods rely on raw *pixels* or manually calculated descriptors, such as the *Histogram of Oriented Gradients* (*HOG*), the CNN automatically learns the coefficients of the convolutional filters during the optimization process.

The following codes (consolidated in Figure 9.15) carry out data preparation, define a simple convolutional architecture in **PyTorch**, execute the training loop using the *Adam* algorithm, and generate the evolution curves of the loss function and accuracy.

Block 1: Data Preparation and Structuring

The initial stage of any Deep Learning *pipeline* consists of converting and adapting the input data to the format required by the scientific computing *framework*.

The Concept of *Tensor*

In Deep Learning, the fundamental data structure is the ***tensor***. From a computational perspective, a *tensor* consists of a multidimensional array of numbers generalized to n dimensions:

- A 0th-order tensor is a scalar (a single value).
- A 1st-order tensor is a vector (length).
- A 2nd-order tensor is a matrix (rows and columns).
- A 3rd-order or higher tensor represents a volume or hyper-array of data.

In the context of **PyTorch**, the `torch.Tensor` class extends the functionality of multidimensional numerical arrays (such as those in **NumPy**) by offering support for hardware-accelerated operations via *GPUs* (*Graphics Processing Units*) and support for automatic derivative computation (*autograd*), which is essential for the backpropagation algorithm.

Analysis of the Preprocessing Code

1. **Loading and Normalizing Intensities:** The `load_digits` dataset contains $1,797$ samples of handwritten digits of 8×8 *pixels*, whose original intensities range on the integer scale from 0 to 16. Dividing by 16.0 performs the **normalization** of the data to the range $[0.0, 1.0]$. This floating-point scaling transformation (`float32`) is indispensable in neural networks to prevent saturation of activation functions and to stabilize the computation of gradients in the optimization algorithm.

2. **Stratified Split (70% Training / 30% Testing):** The `train_test_split` function separates 70% of the samples for adjusting the network parameters and reserves 30% for evaluating the model on unseen data. The `stratify=y` parameter ensures stratified sampling, maintaining the exact proportion of each of the 10 digit classes (0 to 9) in both sets, thereby preventing distribution bias.

3. **Dimensional Adaptation for 2D Convolution (`unsqueeze`):** In CNNs, two-dimensional convolutional layers (`nn.Conv2d`) require that the input *tensor* strictly possess 4 dimensions in the ordering (N, C, H, W):

   - N: number of samples (*batch size*).
   - C: number of color channels (1 for grayscale, 3 for *RGB*).
   - H: image height in *pixels* (8).
   - W: image width in *pixels* (8).

   Since the original array has a 3D shape of the type $(N, 8, 8)$, the call `.unsqueeze(1)` inserts a singleton dimension specifically at **index 1** (the position reserved for the color channel C), transforming the structure into a 4D *tensor* of shape $(N, 1, 8, 8)$, as required by PyTorch.

4. **Label Conversion (`dtype=torch.long`):** The class labels y are converted into 64-bit integer *tensors* (`torch.long`). This type specification is a requirement of the Cross-Entropy loss function (`nn.CrossEntropyLoss`), which uses non-negative integers as indices to associate the correct class with the network's output *logits*.

> **Attention to Dimensions:** The final structure is represented by the **tensor `(N, 1, 8, 8)`**, where `N` is the number of samples (*batch size*), `1` is the color channel (grayscale), and `8×8` is the spatial resolution of the image in *pixels*.

Figure 9.13 illustrates a sequence of samples from the training set after preprocessing and dimensional adaptation to **PyTorch** *tensors*. In the display step, the call `img.squeeze().numpy()` chains two transformations: the `.squeeze()` method eliminates the redundant singleton dimension of the color channel, reducing the 3D *tensor* of shape `(1, 8, 8)` to a 2D matrix of `(8, 8)`; subsequently, the `.numpy()` method converts the **PyTorch** structure into a native **NumPy** array, a format required by graphical rendering tools such as `mm.show()`.

```
# 1. Data loading and preprocessing
digits = load_digits()
X = digits.images.astype(np.float32) / 16.0  # Normalization to the range [0, 1]
y = digits.target

# Stratified split into training (70%) and test (30%) sets
X_treino, X_teste, y_treino, y_teste = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Adjusting to the dimension expected by PyTorch: (N_samples, Channels, Height, Width)
X_treino_t = torch.tensor(X_treino).unsqueeze(1)   # Dimension: (N, 1, 8, 8)
y_treino_t = torch.tensor(y_treino, dtype=torch.long)
X_teste_t = torch.tensor(X_teste).unsqueeze(1)
y_teste_t = torch.tensor(y_teste, dtype=torch.long)

# Display a sample
n_amostras = 8
imgs = [img.squeeze().numpy() for img in X_treino_t[:n_amostras]]
imgs_titles = [str(label.item()) for label in y_treino_t[:n_amostras]]
mm.show(imgs, titles=imgs_titles, cols=n_amostras, figsize=(12, 2.5))
```

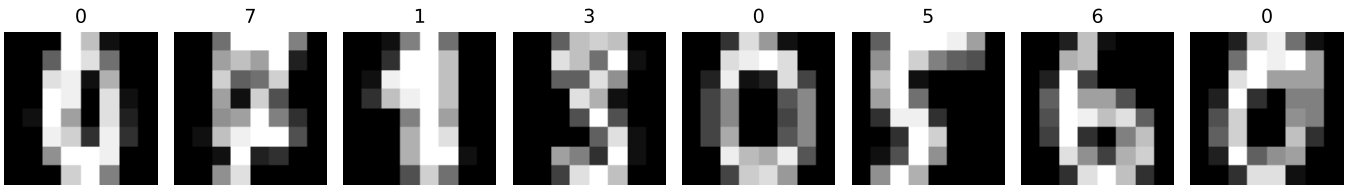


Figure 9.13: Digit samples from the training set after conversion to PyTorch *tensors* and normalization.

Block 2: Definition of the Convolutional Architecture

Model construction in **PyTorch** is structured around the object-oriented paradigm, with a specific class created to represent the neural network (in this example, the `CNNDigitos` class), which inherits all the functionality of the base class `nn.Module`. The `__init__` constructor is responsible for instantiating the layers and declaring their trainable parameters, while the `forward` method establishes the numerical sequence of the forward pass.

Figure 9.14 synthesizes the spatial transformations of the tensors and the data flow throughout the `CNNDigitos` class.

Figure 9.14: Representation of the flow of dimensional transformations of the tensors throughout the *CNNDigitos* architecture.

1. **Constructor (`__init__`) and Component Instantiation:**
   - **Convolutional Layer 1 (`self.conv1`):** Applies 8 filters of size 3×3 with `padding=1` to the grayscale input (1 channel), preserving the spatial resolution of 8×8 pixels.
   - **Convolutional Layer 2 (`self.conv2`):** Processes the 8 feature maps received from the previous layer by applying 16 filters of size 3×3 with `padding=1`.
   - **Subsampling (`self.pool`):** Instantiates the *Max-Pooling* operation with a 2×2 window and stride of 2, reducing the spatial dimension (height and width) by half with each application.
   - **Fully Connected Layers (`self.fc1` and `self.fc2`):** The first dense projection receives the flattened tensor of dimension $16 \times 2 \times 2 = 64$ and produces 32 intermediate features. The second projects these 32 features onto the 10 final output logits.

2. **Forward Propagation in the `forward` Method:**
   - **First Convolutional Block:** The input tensor of shape $(N, 1, 8, 8)$ passes through `conv1` + ReLU and is subsampled by `pool`, resulting in the shape $(N, 8, 4, 4)$.
   - **Second Convolutional Block:** The tensor $(N, 8, 4, 4)$ is processed by `conv2` + ReLU and reduced by `pool` to the shape $(N, 16, 2, 2)$.
   - **Flattening:** The method `x.view(x.size(0), -1)` reshapes the 3D structure into a 1D vector of 64 elements per sample, preserving the batch dimension N.
   - **Classification:** The vector of 64 elements feeds `fc1` with ReLU activation (32 neurons) and ends at `fc2`, producing the 10 unnormalized logits for the computation of the loss function.

```python
# 2. Definition of the Convolutional Architecture
class CNNDigitos(nn.Module):
    """
    Compact convolutional architecture:
    2 convolutional layers with ReLU and Max-Pooling + 2 dense layers.
    """
    def __init__(self, n_classes=10):
        super().__init__()

        # Conv1: 1 input channel, 8 3x3 filters with padding 1 (output: 8x8)
        self.conv1 = nn.Conv2d(1, 8, kernel_size=3, padding=1)
        # Conv2: 8 input channels, 16 3x3 filters with padding 1 (output: 4x4)
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=1)

        self.relu = nn.ReLU()

        # 2x2 Max-Pooling with stride 2
        self.pool = nn.MaxPool2d(2, 2)

        # Fully connected layers (FC)
        self.fc1 = nn.Linear(16 * 2 * 2, 32)
        self.fc2 = nn.Linear(32, n_classes)

    def forward(self, x):
        # First block: Conv (8x8) -> ReLU -> Pool (4x4)
        x = self.pool(self.relu(self.conv1(x)))

        # Second block: Conv (4x4) -> ReLU -> Pool (2x2)
        x = self.pool(self.relu(self.conv2(x)))

        # Flatten: reshapes the 3D matrix (16, 2, 2) into a 1D vector (64)
        x = x.view(x.size(0), -1)

        # Intermediate dense layer with ReLU
        x = self.relu(self.fc1(x))

        # Final classification layer (logits)
        return self.fc2(x)
```

Analysis of Layers and Flow of the `CNNDigitos` Class

1. **Constructor (`__init__`) and Component Instantiation:**
   - **Convolutional Layer 1 (`self.conv1`):** Applies 8 filters of size 3×3 with `padding=1` to the grayscale input (1 channel), preserving the resolution of 8×8 *pixels*.
   - **Convolutional Layer 2 (`self.conv2`):** Processes the 8 received feature maps by applying 16 filters of size 3×3 with `padding=1`.
   - **Subsampling (`self.pool`):** Instantiates the *Max-Pooling* operation with a 2×2 window and stride 2, halving the spatial dimensions (height and width) at each application.
   - **Fully Connected Layers (`self.fc1` and `self.fc2`):** The first dense projection receives the

flattened *tensor* of dimension $16 \times 2 \times 2 = 64$ and produces 32 intermediate features. The second projects these 32 features onto the 10 output *logits*.

2. **Forward Propagation in the `forward` Method:**
   - **First Block:** The *tensor* $(N, 1, 8, 8)$ passes through `conv1` + ReLU and is reduced by `pool` to $(N, 8, 4, 4)$.
   - **Second Block:** The *tensor* $(N, 8, 4, 4)$ passes through `conv2` + ReLU and is reduced by `pool` to $(N, 16, 2, 2)$.
   - **Flattening:** The method `x.view(x.size(0), -1)` converts the 3D structure into a 1D vector of 64 elements per sample.
   - **Classification:** The vector of 64 elements feeds `fc1` with ReLU activation (32 neurons) and ends in `fc2`, which produces the 10 final *logits* for the computation of the Cross-Entropy loss.

Block 3: Instantiation and Optimization Parameters

The learning configuration stage requires instantiating the defined architecture and selecting two core components: the **loss function**, which quantifies the model's error, and the **optimization algorithm**, responsible for adjusting the parameters toward the minimum of that function.

1. **Instantiation and Parameter Counting:** The model is created by instantiating the `modelo_cnn` object from the `CNNDigitos` class. The expression `sum(p.numel() for p in modelo_cnn.parameters())` iterates over all trainable parameter *tensors* of the network (weights and biases of each layer) and calculates the total cardinality of the model, quantifying its representational capacity.

2. **Loss Function (`nn.CrossEntropyLoss`):** Cross-Entropy Loss is the standard choice for multiclass classification problems. In PyTorch, this implementation internally combines the application of the *LogSoftmax* function with Negative Log-Likelihood Loss (*NLLLoss*). For this reason, the network's output layer produces raw logits, eliminating the need for an explicit *Softmax* function at the end of the `forward` method.

3. **Adaptive Optimizer (`optim.Adam`):** Parameter updates use the *Adam* algorithm (*Adaptive Moment Estimation*), with an initial learning rate of $\eta = 0.01$ (`lr=1e-2`). *Adam* combines the principles of momentum with step-size adaptation based on moving averages of first- and second-order derivatives, individually adjusting the learning rate for each network parameter.

```
# 3. Model Initialization and Optimization Parameters
modelo_cnn = CNNDigitos()
num_params = sum(p.numel() for p in modelo_cnn.parameters())
print(f"Trainable model parameters: {num_params}")

criterio = nn.CrossEntropyLoss()
otimizador = optim.Adam(modelo_cnn.parameters(), lr=1e-2)
```

```
Trainable model parameters: 3658
```

Block 4: Training Loop and Evaluation

CNN training occurs iteratively through the Stochastic Gradient Descent algorithm by *mini-batches* (*Mini-batch SGD*).

The learning curves resulting from this process are shown in Figure 9.15, generated at the end of execution.

1. **Training Phase (`modelo_cnn.train()`):** The main loop executes training over 50 epochs. In each epoch, the following steps occur:
   - **Stochastic Shuffling:** The function `torch.randperm(n)` generates a random permutation of the sample indices, ensuring that the ordering of *mini-batches* varies each epoch to avoid sampling biases.

- **Division into *Mini-batches*:** The training set is sliced into batches of 32 samples (`tam_lote = 32`).
- **Zeroing Gradients (`otimizador.zero_grad()`):** Clears the gradients accumulated in the *tensor* from the previous iteration, preventing the unwanted summation of derivatives across distinct batches.
- **Forward Pass and Loss:** The *forward pass* computes the predictions `saida`, and the call `criterio(saida, y_treino_t[idx])` quantifies the batch error.
- **Backpropagation (`perda.backward()`):** Applies the chain rule to compute the partial derivatives of the loss with respect to each parameter ($\frac{\partial L}{\partial w}$).
- **Weight Update (`otimizador.step()`):** Updates the model parameters according to the *Adam* optimizer equations.

2. **Evaluation Phase (`modelo_cnn.eval()`):** At the end of each epoch, the model is switched to evaluation mode. The context `with torch.no_grad()` temporarily disables the automatic derivative computation engine (*autograd*), reducing memory consumption and accelerating inference on the test set (`X_teste_t`). The operation `.argmax(dim=1)` extracts the class with the highest probability for each sample, allowing the test accuracy to be calculated.

3. **Visualization with the `morph` Library:** The function `mm.showTrainCurves` from the didactic library `morph` consolidates the training loss history and test accuracy into a single graphical panel, allowing the model convergence to be diagnosed and the stability of learning to be monitored across epochs.

```python
# 4. Training Loop (Mini-batch SGD)
n = X_treino_t.size(0)                    # Number of samples
tam_lote = 32                             # Mini-batch size
epocas = 50                               # Total epochs
historico_perda, historico_acc = [], []   # Metrics history

for epoca in range(epocas):                  # Repeat for each epoch
    modelo_cnn.train()                       # Training mode
    perm = torch.randperm(n)                 # Shuffle samples
    perda_epoca = 0.0                        # Accumulate losses
    for i in range(0, n, tam_lote):          # Iterate over mini-batches
        idx = perm[i:i + tam_lote]           # Batch indices
        otimizador.zero_grad()               # Zero gradients
        saida = modelo_cnn(X_treino_t[idx])  # Forward pass
        perda = criterio(saida, y_treino_t[idx]) # Compute loss
        perda.backward()                         # Backpropagation
        otimizador.step()                        # Update weights
        perda_epoca += perda.item() * len(idx)   # Sum loss

    # Model evaluation on the test set at the end of each epoch
    modelo_cnn.eval()                            # Evaluation mode
    with torch.no_grad():                        # No gradients
        pred_teste = modelo_cnn(X_teste_t).argmax(dim=1)  # Predictions
        acc_teste = (pred_teste == y_teste_t).float().mean().item()  # Accuracy
    historico_perda.append(perda_epoca / n)      # Record loss
    historico_acc.append(acc_teste)              # Record accuracy

acc_final_cnn = historico_acc[-1]                # Final accuracy
print(f"Final accuracy of the CNN on the test set: {acc_final_cnn:.4f}")  # Display result

final = mm.showTrainCurves(                      # Plot curves
    historico_perda, historico_acc,
    titulo="Evolução do Treinamento da CNN - Base de Dígitos",
    subtitulo=f"Acurácia final no teste: {acc_final_cnn:.4f}",
)
```

```
Final accuracy of the CNN on the test set: 0.9759
```

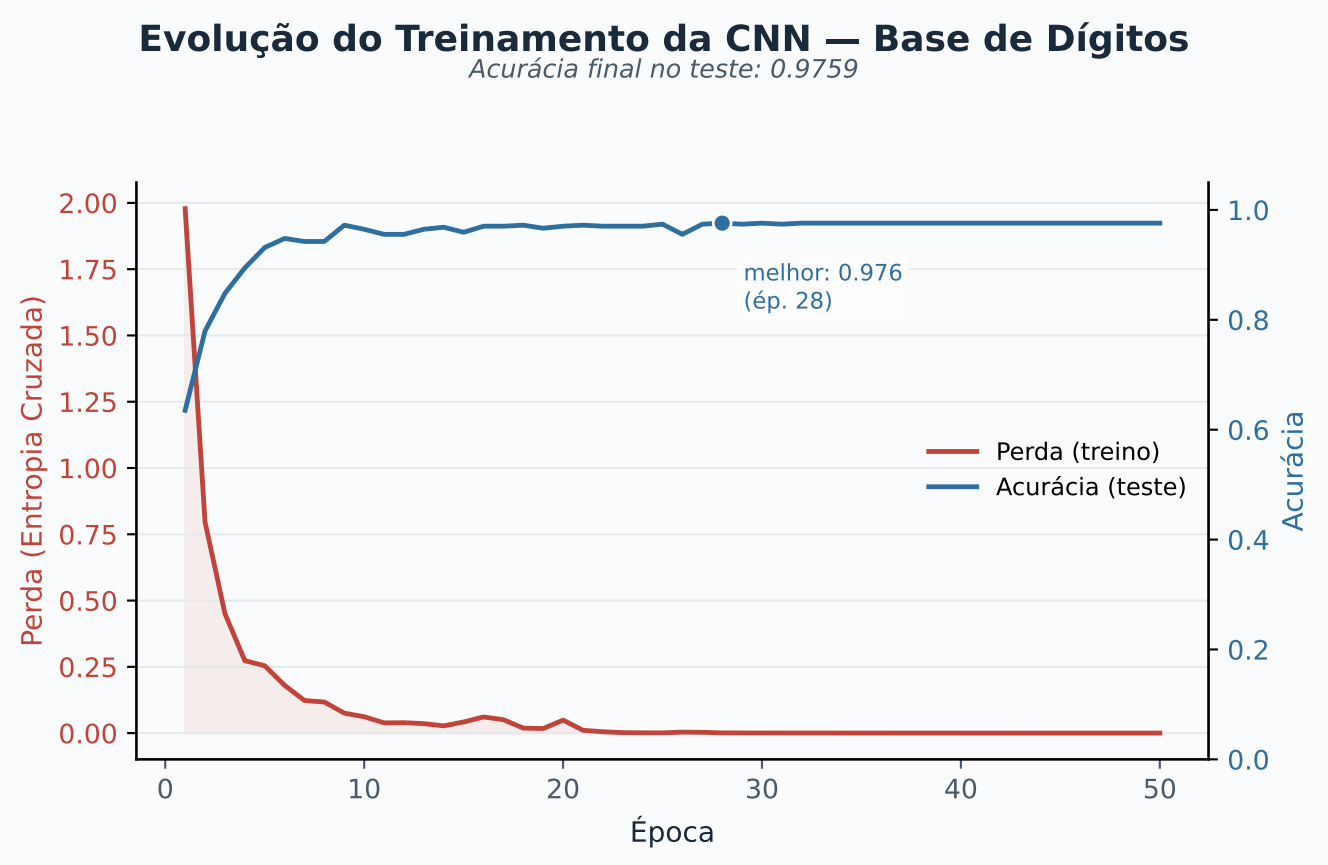


Figure 9.15: Training and evaluation curves of the CNN on the digits dataset: evolution of cross-entropy loss on the training set and accuracy on the test set over 50 epochs.

Block 5: Visualization of the Activation Flow

Inspecting the trained network allows observing the progressive transformation of the input *tensor* across the layers of the `CNNDigitos` architecture. Figure 9.16 illustrates the dimensions and intermediate activations obtained when processing a real example of the digit 3.

1. **Sample Selection and Preparation:** The stochastic seed is fixed with `torch.manual_seed(7)` to ensure reproducibility of the results. The first occurrence of the digit 3 in the `load_digits` dataset is isolated, normalized to the interval $[0.0, 1.0]$, and reshaped as a *tensor* `x` of dimension $(1, 1, 8, 8)$.

2. **Intermediate Inspection with `mm.showNet`:** The `mm.showNet` function from the `morph` library performs the *forward pass* of the *tensor* `x` through the previously trained `modelo_cnn` instance. Using *forward hooks*, the function intercepts the numerical state of activations in the convolutional (`nn.Conv2d`), pooling (`nn.MaxPool2d`), and fully connected (`nn.Linear`) layers, returning them in the `acts` dictionary. The nonlinear activation functions (`nn.ReLU`) are not registered as independent stages, since their application occurs directly on the output *tensor* of the corresponding layer.

3. **Result Verification:** The command `list(acts.keys())` displays the sequence of identifiers for the monitored layers, allowing confirmation of the progressive dimensional reduction and the generation of the *logit* with maximum value at the index corresponding to class 3, as demonstrated in Figure 9.16.

```
torch.manual_seed(7)
digits = load_digits()
idx = np.where(digits.target == 3)[0][0]
img = digits.images[idx] / 16.0
x = torch.tensor(img, dtype=torch.float32).view(1, 1, 8, 8)

# Reuse of the previously trained model instance
```

```
acts = mm.showNet(
    modelo_cnn,
    x,
    titulo="Fluxo de transformações dos tensors ao longo da arquitetura CNNDigitos",
    subtitulo=f"Exemplo real do dataset load_digits (classe verdadeira:
{digits.target[idx]})",
)
print("Captured layers:", list(acts.keys()))
```

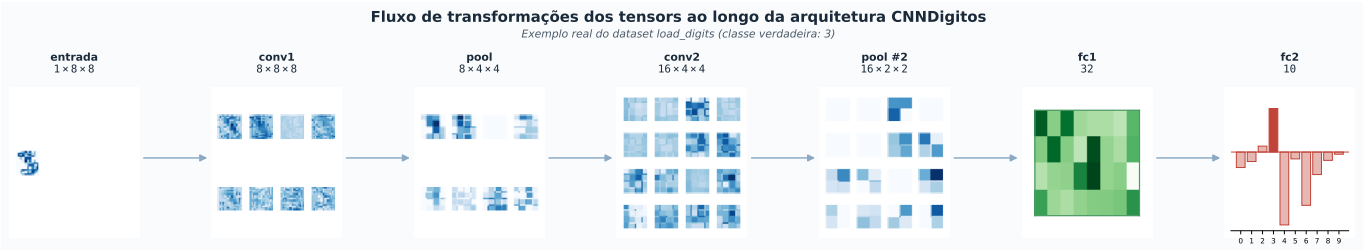


Figure 9.16: Flow of activations of the trained CNN when processing a real example of the digit 3, from the *load_digits* dataset: tensor dimensions layer by layer, from input to output *logit*.

```
Captured layers: ['conv1', 'pool', 'conv2', 'pool #2', 'fc1', 'fc2']
```

Inspecting the Computational Graph with `torchviz`

While `mm.showNet` prioritizes didactic clarity—displaying one column per layer with trainable parameters—the `torchviz` library projects the **autograd graph** exactly as PyTorch builds it internally for gradient computation. Figure 9.17 illustrates this perspective by representing the `CNNDigitos` architecture.

1. **Tracked Forward Propagation:** With the trained model in `eval()` mode, the *forward pass* over the `x` tensor of digit 3 is sufficient for the *autograd* engine to record all executed operations, including those without trainable parameters, such as the `ReLU` activation function and the `view` dimensional reshaping.

2. **Graph Generation (`make_dot`):** The `make_dot(saida, params=...)` function builds the graph from the output tensor by traversing backward through the operation history to the leaf nodes (the model's trainable parameters). Each node in the diagram represents an operation of the *backward pass* (such as `ReluBackward` or `AddmmBackward`), not merely a conceptual block of the `nn.Module`.

3. **Export and Rendering (`.render`):** The `.render(..., format="png", cleanup=True)` method invokes the `dot` executable from **Graphviz** to compile the image in PNG format, automatically removing intermediate source-code files.

Figure 9.17 highlights how this computational graph, even for a compact architecture, exhibits greater density than the `mm.showNet` panel, as it details each atomic operation responsible for the gradient flow.

```
# 1. Forward pass with gradient tracking enabled
modelo_cnn.eval()
saida = modelo_cnn(x)  # Reuse of tensor x (digit 3)

# 2. Basic graph: flow of operations up to the output
grafo_simples = make_dot(saida, params=dict(modelo_cnn.named_parameters()))
caminho_simples = grafo_simples.render("cnn_digitos_grafo_simples", format="png",
cleanup=True)

# 3. Direct display in the Quarto/Jupyter environment
# The .render() returns the path of the generated PNG file; we need to open it as an image
imagem_simples = np.array(Image.open(caminho_simples).convert("RGB"))
```

```
mm.show(imagem_simples, figsize=(5,10))
```

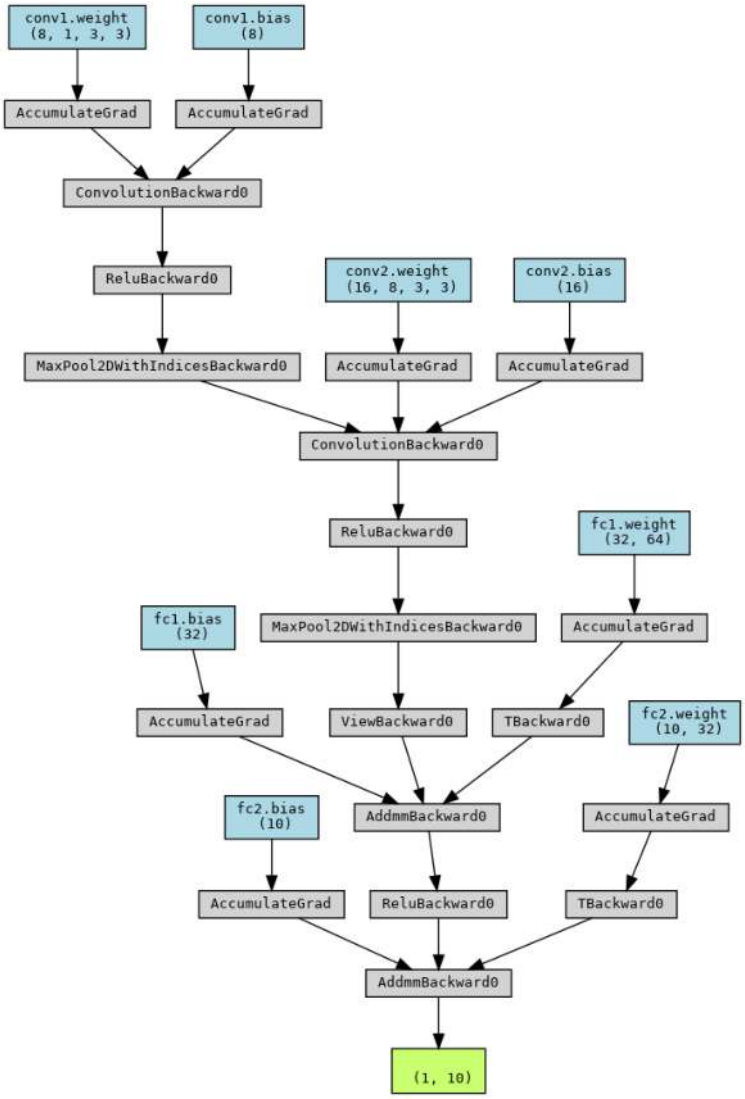


Figure 9.17: Computational graph of *CNNDigitos* generated via *torchviz*, showing the *forward* operations and the gradient nodes (*backward*) associated with each trainable parameter.

Detailed View of the Computational Graph with `torchviz`

In addition to the simplified representation, the `torchviz` library allows expanding the *autograd* graph to inspect the internal execution details of the `CNNDigitos` network. Figure 9.18 presents this expanded structure for the same input *tensor* `x`.

1. **Tracing with Operation Attributes (`show_attrs=True`):** The inclusion of attributes displays the hyperparameter configurations associated with each computational node during the forward pass, such as kernel dimensions (`kernel_size`), strides, and paddings in convolutions and subsamplings.

2. **Detection of Saved Tensors in Memory (`show_saved=True`):** This parameter forces the explicit display of the intermediate tensors that PyTorch retains in memory during the forward pass. These data are preserved because they will be strictly necessary for calculating the partial derivatives during the backward pass.

3. **Generation and Compilation of Graphs:** While `grafo_simples` generates a direct view of the gradient flow, `grafo_detalhado` compiles the expanded graph into the file

`cnn_digitos_grafo_detalhado.png` via the `dot` executable from **Graphviz**.

As observed in Figure 9.18, this detailed visualization is useful for debugging video memory (VRAM) consumption and verifying how the PyTorch engine internally allocates each node of the chain rule.

```
# same code as before (make_dot + render)...

# 2. Detailed graph: display of dimensions and tensors saved for backward
grafo_detalhado = make_dot(
    saida,
    params=dict(modelo_cnn.named_parameters()),
    show_attrs=True,   # Displays operation attributes (e.g., kernel_size, stride)
    show_saved=True,   # Displays tensors saved in memory for backpropagation
)

caminho_detalhado = grafo_detalhado.render("cnn_digitos_grafo_detalhado",
                                           format="png", cleanup=True)

# 3. Direct display in the Quarto/Jupyter environment
imagem_detalhado = np.array(Image.open(caminho_detalhado).convert("RGB"))
mm.show(imagem_detalhado, figsize=(8, 16))
```

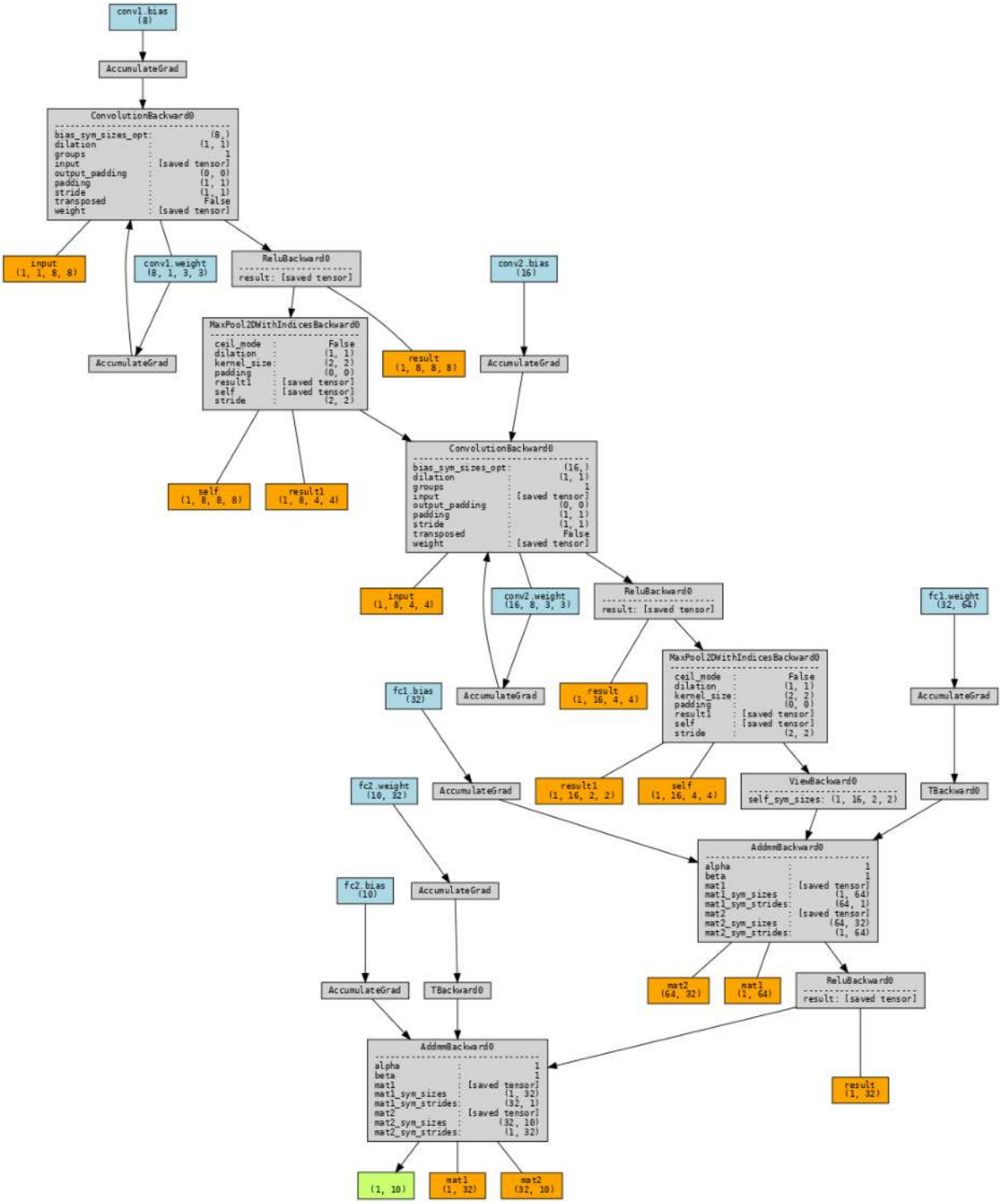


Figure 9.18: Detailed graph of the CNNDigitos class generated via *torchviz*.

Comparison with Chapter 7

Figure 9.19 brings together the results obtained on the same dataset (`load_digits`), establishing a direct parallel between the classical approaches explored previously and the CNN developed in this chapter.

1. **Performance of Raw Pixels vs. Manual Descriptors:** In the experiments from Chapter 7, the k-NN classifier ($k = 3$) achieved an accuracy of 98.4% when fed directly with the raw pixels of the images. In contrast, prior feature extraction via the Histogram of Oriented Gradients (HOG) resulted in significantly lower performance (75.8%). This decline occurs because HOG was designed to capture edge gradients in higher-resolution images; in matrices of only 8×8 pixels, the spatial resolution is insufficient to form informative orientation histograms.

2. **Equivalence of CNN and End-to-End Learning:** The convolutional network `CNNDigitos` achieves competitive performance of 97.6%, approaching the accuracy of k-NN with raw pixels on a small and pre-aligned dataset. The major conceptual advantage lies in representation learning: instead of relying on manually designed descriptors (handcrafted features) or keeping the entire dataset in memory for neighbor searching at inference time, the CNN automatically optimizes its own convolutional filters during training, yielding a compact model capable of performing feature extraction and classification in an integrated (end-to-end) manner.

```
import matplotlib.pyplot as plt

# Values obtained in Chapter 7 (k-NN, k=3), reproduced for direct comparison
ACC_KNN_PIXELS_CAP7 = 0.9844
ACC_KNN_HOG_CAP7 = 0.7578

metodos = ["k-NN\n(pixels brutos)", "k-NN\n(HOG)", "CNN\n(este capítulo)"]
acuracias = [ACC_KNN_PIXELS_CAP7, ACC_KNN_HOG_CAP7, acc_final_cnn]

plt.figure(figsize=(5, 4))
cores = ["#6366f1", "#f97316", "#16a34a"]
plt.bar(metodos, acuracias, color=cores)
plt.ylim(0, max(acuracias) + 0.08)
plt.ylabel("Acurácia (conjunto de teste)")
plt.title("Cap. 7 vs. Cap. 9 - Base de Dígitos")

for i, v in enumerate(acuracias):
    plt.text(i, v + 0.02, f"{v:.3f}", ha="center")

plt.tight_layout()
plt.show()
```

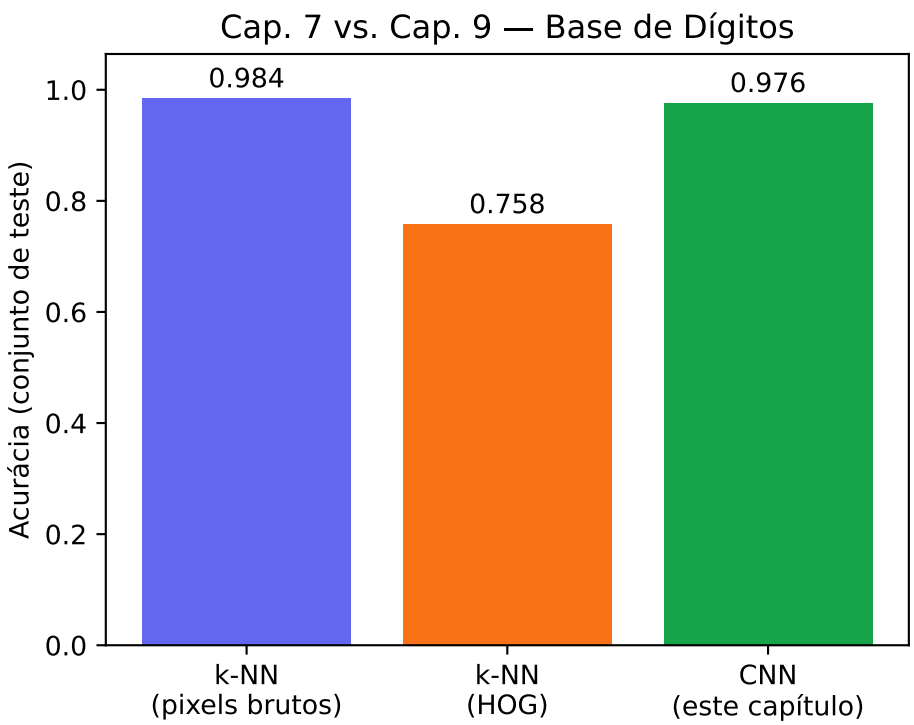


Figure 9.19: Comparison of accuracy between the classic classifiers from Chapter 7 (raw pixels and HOG with k-NN) and the CNN trained in this chapter, on the same digit dataset.

🧠 Why It Works — And Why the CNN Does Not Always "Win"

The observed result here **repeats the pattern already seen in Chapter 7**: the CNN, despite automatically learning its features, does not necessarily outperform the k-NN with raw *pixels* on this specific dataset. The explanation is the same: `load_digits` is a small dataset (fewer than $1,800$ examples), with images that are already centered, normalized, and of very low resolution (8×8) — conditions under which direct comparison of intensities is already highly informative, and there is little data for the network to learn filters that are truly superior to simple descriptors.
The real differentiator of CNNs appears in scenarios that handcrafted descriptors and simple classifiers cannot address: larger and more realistic images, with thousands of categories, substantial variation in pose, lighting, and background, and massive training sets — exactly the regime in which the models presented in the section *"Large-Scale Applications"*, later on, were trained. The pedagogical lesson that runs through Chapters 7, 8, and 9 of this book is consistent: **the sophistication of a method should be proportional to the complexity of the problem** — using a CNN for a problem that a k-NN solves equally well is a waste of computational resources, not a virtue.
This same proportionality applies to the inspection tools used throughout the chapter. The `mm.showNet` was built for **didactic** purposes and works well on shallow networks such as `CNNDigitos`, but it does not scale to deep architectures: each traced layer becomes a column in the figure, and convolutional layers with hundreds of channels generate mosaics too large for visual interpretation; furthermore, the *hooks* store all activations in memory, and the *layout* assumes a sequential flow, not faithfully representing residual connections or branches (as in *ResNets* or *Inception* modules). Thus, `showNet` should be understood as a pedagogical lens for small networks — analogous to the role of `mm.showBoundBox` in the visual debugging of detections — and not as a substitute for production-oriented tools such as *TensorBoard* or *torchviz*.

9.5.1.2 Transfer Learning

Training a CNN from scratch generally requires a large amount of labeled data and significant computational resources, as the training process must adjust all network parameters. In many applications, however, only a reduced set of data is available for the task of interest. In this situation, **transfer learning** reuses the representations learned by a model previously trained on a source task with a large volume of data, reducing training costs and the need for new samples.

In CV, this strategy explores the hierarchical organization of CNNs. The initial layers learn low-level visual features, such as edges, textures, intensity gradients, and color patterns, which remain useful across different domains. The deeper layers combine this information to form progressively more abstract and specialized representations, related to the classes present in the training base.

This section investigates under what conditions transfer learning yields good results. The first experiment shows that a small extractor trained on a restricted domain can lead to **negative transfer**. The second demonstrates why deep models pre-trained on large image datasets achieve high performance on new tasks. Finally, the third applies this strategy to a plant health diagnosis problem, illustrating a scenario close to real-world applications.

Experiment 1 — Limitations of a Small and Specialized Extractor

The first experiment shows that transfer learning does not always improve a model's performance. To this end, the handwritten digits dataset (`load_digits`) is divided into two disjoint domains:

- **Domain A (source):** digits 0 to 4, used to train a small CNN;
- **Domain B (target):** digits 5 to 9, reindexed to 0 to 4, forming a new task with only 20 training samples.

The objective is to evaluate the effect of reusing the feature extractor learned in Domain A without allowing its adaptation to Domain B.

Block 1: Division of Domains, Scarcity, and Display of Samples

This block prepares the dataset for the experiment. Unlike the previous project, which used all digits in a single classification problem, the dataset is divided into two independent tasks: a source task (Domain A) and a target task (Domain B).

Figure 9.20 presents examples from both domains after class separation, conversion to **PyTorch** *tensors*, and preprocessing.

1. **Class separation:** The boolean masks `mask_A` and `mask_B` separate the examples from each domain. Next, the code reindexes the labels of Domain B (`y[mask_B] - 5`) to the range $[0, 4]$, allowing both models to use five output classes.

2. **Data scarcity:** The generator `np.random.default_rng(0)` selects only 20 samples for training Domain B, approximately four per class, simulating a scenario where training from scratch tends to suffer from *overfitting*.

3. **Conversion to tensors:** The `para_tensor` function converts the images to the format $(N, 1, 8, 8)$ and the labels to `torch.long`, compatible with the `nn.Conv2d` layers and the loss function.

4. **Sample visualization:** The code uses `.squeeze().numpy()` to convert the tensors into **NumPy** arrays. Figure 9.20 presents examples from both domains and highlights the reindexing applied to the labels of Domain B.

```
# 1. Splitting the dataset into two disjoint domains
classes_A, classes_B = [0, 1, 2, 3, 4], [5, 6, 7, 8, 9]
mask_A, mask_B = np.isin(y, classes_A), np.isin(y, classes_B)

XA, yA = X[mask_A], y[mask_A]
XB, yB = X[mask_B], y[mask_B] - 5  # Reindexing the labels to the range [0, 4]

# Splitting into training and test sets for both domains
XA_tr, XA_te, yA_tr, yA_te = train_test_split(
    XA, yA, test_size=0.25, random_state=42, stratify=yA
)
XB_tr, XB_te, yB_tr, yB_te = train_test_split(
    XB, yB, test_size=0.25, random_state=42, stratify=yB
)

# Simulation of extreme scarcity in the target domain: only 20 training samples
rng = np.random.default_rng(0)
idx_poucos = rng.choice(len(XB_tr), size=20, replace=False)
XB_tr_poucos, yB_tr_poucos = XB_tr[idx_poucos], yB_tr[idx_poucos]

# Helper function for conversion into PyTorch tensors
def para_tensor(Ximg, yarr):
    return torch.tensor(Ximg).unsqueeze(1), torch.tensor(yarr, dtype=torch.long)

XA_tr_t, yA_tr_t = para_tensor(XA_tr, yA_tr)
XA_te_t, yA_te_t = para_tensor(XA_te, yA_te)
XB_tr_t, yB_tr_t = para_tensor(XB_tr_poucos, yB_tr_poucos)
XB_te_t, yB_te_t = para_tensor(XB_te, yB_te)

# Display of samples from both domains
n_amostras = 5
imgs_A = [img.squeeze().numpy() for img in XA_tr_t[:n_amostras]]
titles_A = [f"A: {label.item()}" for label in yA_tr_t[:n_amostras]]

imgs_B = [img.squeeze().numpy() for img in XB_tr_t[:n_amostras]]
titles_B = [f"B: {label.item()} (orig: {label.item()+5})" for label in yB_tr_t[:n_amostras]]
```

```
mm.show(
    imgs_A + imgs_B,
    titles=titles_A + titles_B,
    cols=n_amostras,
    figsize=(12, 4.5)
)
```

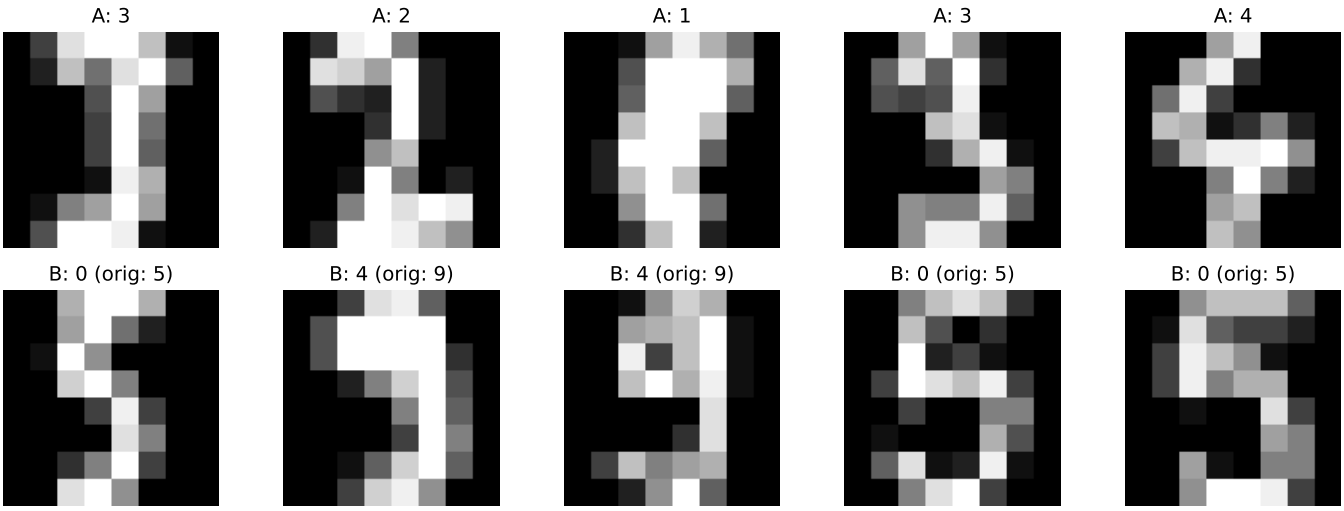


Figure 9.20: Samples from the training sets after preprocessing and dimensionality adjustment to PyTorch tensors: Domain A (digits 0 to 4, source task) and Domain B (digits 5 to 9 reindexed to 0 to 4, target task).

Block 2: Modular Architecture and Generic Routines

To enable learning transfer, the convolutional architecture and training loop were refactored relative to the `CNNDigitos` class from the previous project.

1. **Architecture Modularization (Difference from `CNNDigitos`):**
   - In the previous project, the `CNNDigitos` class declared all layers (`conv1`, `conv2`, `pool`, `fc1`, `fc2`) as direct members of a single monolithic class.
   - Here, the architecture is separated into two components: the `ExtratorConv` class encapsulates the convolutional spatial block (2 3×3 convolutions, 2 2×2 Max-Poolings, and flattening to 64 elements), while the `CNNCompleta` class instantiates this extractor in `self.extrator` and attaches the classifier "head" (`fc1` and `fc2`).
   - This separation is what allows the internal state of the extractor (`state_dict()`) to be copied from one model to another in an isolated manner.
2. **Adjustment in the Number of Output Classes:** While `CNNDigitos` in the previous project had 10 logits in the output layer (`self.fc2 = nn.Linear(32, 10)`), the `CNNCompleta` class receives `n_classes=5` in the constructor to suit the division of domains A and B.
3. **Flexibilization of the Training Loop (`treinar`):**
   - In the previous project, the training loop iterated directly over the model's global attributes (`modelo_cnn.parameters()`) and computed specific metrics inline.
   - The `treinar` function abstracts this process and introduces the optional parameter `parametros`. If provided, the *Adam* optimizer updates **only** the parameters in that list, ignoring layers whose gradients have been deactivated. This flexibility is crucial for performing training with partial freezing of the network.
4. **Isolation of Evaluation (`calcular_acuracia`):** As done in the testing phase of the previous project, the function sets the model to `eval()` and uses the `torch.no_grad()` context to deactivate autograd, computing accuracy via `.argmax(dim=1)`.

```python
# Definition of the reusable convolutional block (same extraction as the previous project)
class ExtratorConv(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 8, 3, padding=1)
        self.conv2 = nn.Conv2d(8, 16, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        return x.view(x.size(0), -1)

# Modular architecture combining the extractor and the classification head
class CNNCompleta(nn.Module):
    def __init__(self, n_classes=5):
        super().__init__()
        self.extrator = ExtratorConv()
        self.fc1 = nn.Linear(64, 32)
        self.fc2 = nn.Linear(32, n_classes)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.extrator(x)
        x = self.relu(self.fc1(x))
        return self.fc2(x)

# Generic training routine with selective parameter optimization
def treinar(modelo, X_t, y_t, epocas, lr, tam_lote=16, parametros=None):
    # Trainable parameters
    params = parametros if parametros is not None else modelo.parameters()
    otim = optim.Adam(params, lr=lr)                # Adam optimizer
    crit = nn.CrossEntropyLoss()                    # Loss function
    n_amostras = X_t.size(0)                        # Number of samples
    for _ in range(epocas):                         # Repeat per epoch
        perm = torch.randperm(n_amostras)           # Shuffles samples
        for i in range(0, n_amostras, tam_lote):    # Iterates over mini-batches
            idx = perm[i:i + tam_lote]              # Batch indices
            otim.zero_grad()                        # Zeros gradients
            perda = crit(modelo(X_t[idx]), y_t[idx])  # Computes loss
            perda.backward()                        # Backpropagation
            otim.step()                             # Updates weights

# Evaluation routine
def calcular_acuracia(modelo, X_t, y_t):
    modelo.eval()                                   # Evaluation mode
    with torch.no_grad():                           # No gradients
        pred = modelo(X_t).argmax(dim=1)            # Predicted classes
    return (pred == y_t).float().mean().item()      # Returns accuracy
```

Block 3: Pre-training, Transfer, and Comparative Analysis

This block carries out a comparison between training the model from scratch and applying transfer learning with static freezing of the feature extractor. To ensure full transparency of the experiment, the sizes of the training and test sets for both domains are printed to the terminal.

1. **Quantification of Samples per Domain:**

    - **Domain A (source, digits 0 to 4):** It comprises 675 training samples (75%) and 226 test samples (25%), providing abundant data for the `source_model` to learn the convolutional feature

extractor until it reaches 100% accuracy.

- **Domain B (target, digits 5 to 9):** It has 224 test samples in total, but its training set is intentionally reduced from 672 to only 20 **samples** (`XB_tr_few`), creating a severe scenario of data scarcity.

2. **Step 1: Pre-training on Domain A (Source):** The `source_model` is trained from scratch on the 675 samples of digits 0 to 4. Over 40 epochs, the convolutional feature extractor adjusts its filters to identify the distinctive features of these first five digits, reaching 100% accuracy on the test set (226 samples).

3. **Step 2: Weight Transfer and Freezing:**
   - The `transfer_model` is created to solve the Domain B task (digits 5 to 9).
   - The weights learned on Domain A are copied via:
     - `load_state_dict(source_model.extractor.state_dict())`
   - **Freezing:** The loop `for p in transfer_model.extractor.parameters(): p.requires_grad = False` disables gradient computation in the convolutional layers.
   - **Selective Training:** The `train(...)` call strictly passes the parameters of the dense layers (`head_params`), adjusting the classification head with only the 20 training samples.

4. **Step 3: Training from Scratch on Domain B (Experimental Control):** The `from_scratch_model` has the same architecture but is trained from scratch on the same 20 samples from Domain B, without any weight reuse, over the same 40 epochs.

5. **Analysis of Results (Figure 9.21):**
   - **With Frozen Transfer (72.77%):** By reusing the feature extractor trained on Domain A and freezing its parameters, the network achieves 72.77% test accuracy (224 samples) by adjusting only the dense layers.
   - **Trained from Scratch (76.79%):** Training from scratch outperforms frozen transfer on the Domain B test set.
   - **Reason for the Difference:** Since it is a tiny model (only 16 convolutional filters on 8×8 matrices), the feature extractor trained on Domain A became **hyperspecialized** in the geometric shapes of digits 0 to 4. By rigidly freezing these few filters, the target model was limited to detectors unsuitable for 5 to 9. The network trained from scratch, even with only 20 samples, was able to adapt its 16 filters directly to the features of Domain B.

```
# Fix seed for reproducibility
torch.manual_seed(42)

# Display training and test group sizes
print("=== Database Size Breakdown ===")
print(f"Domain A (0-4) - Training: {len(XA_tr_t)} samples | Test: {len(XA_te_t)} samples")
print(f"Domain B (5-9) - Full training: {len(XB_tr)} | Reduced training: {len(XB_tr_poucos)}",
      f"| Test: {len(XB_te_t)} samples\n")

# 1. Pre-training on the source task (Domain A: digits 0-4)
modelo_origem = CNNCompleta(n_classes=5)                        # Create CNN

#######
treinar(modelo_origem, XA_tr_t, yA_tr_t, epocas=40, lr=1e-2)   # Train model

acc_A = calcular_acuracia(modelo_origem, XA_te_t, yA_te_t)     # Measure accuracy

print(f"Accuracy in source domain A "                       # Display result
      f"(digits 0-4, {len(XA_te_t)} tests): " f"{acc_A:.4f}"# Display result
      f"(dígitos 0-4, {len(XA_te_t)} testes): " f"{acc_A:.4f}")
```

```
# 2. Transfer Learning (Frozen Extractor)
modelo_transferencia = CNNCompleta(n_classes=5)        # Create CNN
modelo_transferencia.extrator.load_state_dict(         # Copy extractor
    modelo_origem.extrator.state_dict())

for p in modelo_transferencia.extrator.parameters():   # Traverse extractor
    p.requires_grad = False                            # Freeze weights

params_cabeca = list(modelo_transferencia.fc1.parameters())  # FC1
params_cabeca += list(modelo_transferencia.fc2.parameters()) # +FC2

#######
treinar(modelo_transferencia, XB_tr_t, yB_tr_t,              # Train head
         epocas=40, lr=1e-2, parametros=params_cabeca)

acc_transferencia = calcular_acuracia(modelo_transferencia, XB_te_t, yB_te_t) # Measure
accuracy

# 3. Training from Scratch in Domain B
modelo_do_zero = CNNCompleta(n_classes=5)                     # Create CNN

#######
treinar(modelo_do_zero, XB_tr_t, yB_tr_t, epocas=40, lr=1e-2) # Train model

acc_do_zero = calcular_acuracia(modelo_do_zero,  XB_te_t, yB_te_t) # Measure accuracy


print(f"Target domain B (digits 5-9), only {len(XB_tr_poucos)} ",
      f"training examples ({len(XB_te_t)} tests):")
print(f"  With transfer (frozen extractor): {acc_transferencia:.4f}")
print(f"  Training from scratch (same data/epochs): {acc_do_zero:.4f}")

# Comparative visualization
plt.figure(figsize=(4.5, 4))
plt.bar(["Do zero", "Transferência"], [acc_do_zero, acc_transferencia],
        color=["#dc2626", "#16a34a"])
plt.ylim(0, max([acc_do_zero, acc_transferencia]) + 0.1)
plt.ylabel("Acurácia no domínio B (teste)")
plt.title(f"Efeito da Transferência ({len(XB_tr_poucos)} exemplos de treino)")

for i, v in enumerate([acc_do_zero, acc_transferencia]):
    plt.text(i, v + 0.02, f"{v:.3f}", ha="center")

plt.tight_layout()
plt.show()
```

```
=== Database Size Breakdown ===
Domain A (0-4) - Training: 675 samples | Test: 226 samples
Domain B (5-9) - Full training: 672 | Reduced training: 20 | Test: 224 samples

Accuracy in source domain A (digits 0-4, 226 tests): 1.0000(dígitos 0-4, 226 testes): 1.0000
Target domain B (digits 5-9), only 20  training examples (224 tests):
  With transfer (frozen extractor): 0.7277
  Training from scratch (same data/epochs): 0.7679
```

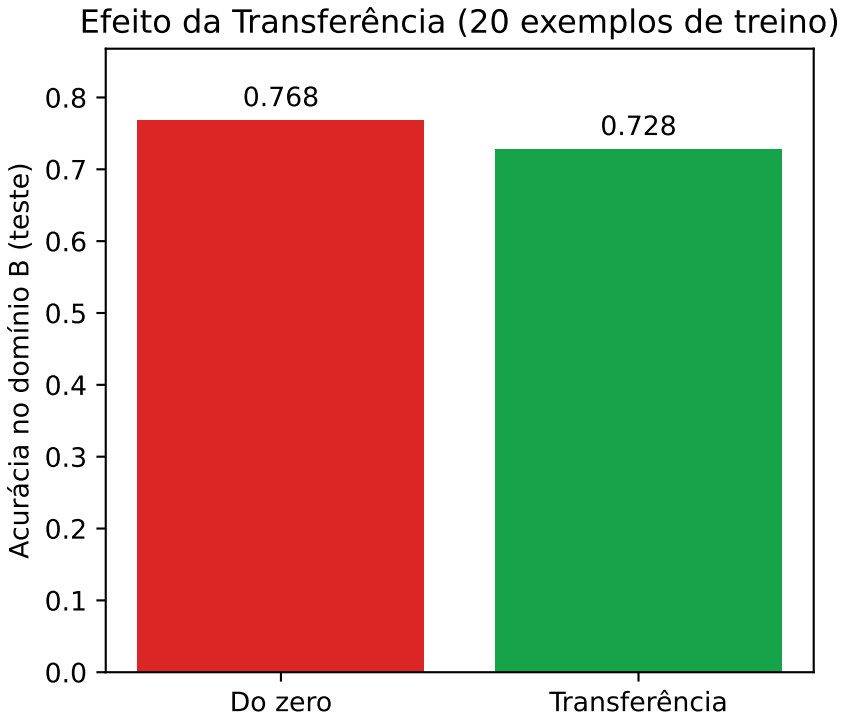


Figure 9.21: Comparison of accuracy on the Domínio B test set (digits 5 to 9) under data restriction (20 training examples): demonstration of the impact of rigid freezing and negative transfer in low-capacity networks.

Block 4: Visualization of the Activation Flow with `mm.showNet`

To confirm that the reused feature extraction preserves the dimensional transformations studied in the previous project, the `mm.showNet` function from the `morph` library is used again. Figure 9.22 displays the activation flow of `modelo_transferencia` when processing a sample from Domain B (digit 7, reindexed to class 2).

1. **Preservation of the Convolutional Flow:** Since the `ExtratorConv` architecture replicates the same convolution and pooling layers as the `CNNDigitos` from the previous project, the dimensions of the intermediate tensors remain at $(1, 8, 4, 4)$ in the first block.
2. **Inspection of the Adapted Head:** The difference from the previous project arises in the output layer (`fc2`): whereas the previous project's model projected the intermediate vector into 10 *logits* (classes 0 through 9), the transfer model projects the vector into 5 *logits* (classes 0 through 4), capturing the relative probabilities of Domain B.

```
# Selects the first test sample from Domain B
x_amostra_B = XB_te_t[0:1]  # Tensor of dimension (1, 1, 8, 8)
classe_verdadeira = yB_te_t[0].item()
classe_original = classe_verdadeira + 5

# Inspection of the activation flow in the transfer model
acts_transfer = mm.showNet(
    modelo_transferencia,
    x_amostra_B,
    titulo="Fluxo de ativações no modelo de transferência (Domínio B)",
    subtitulo=f"Amostra do dígito {classe_original} (rótulo reindexado: {classe_verdadeira})",
)

print("Layers captured in the transfer model:\n", list(acts_transfer.keys()))
```

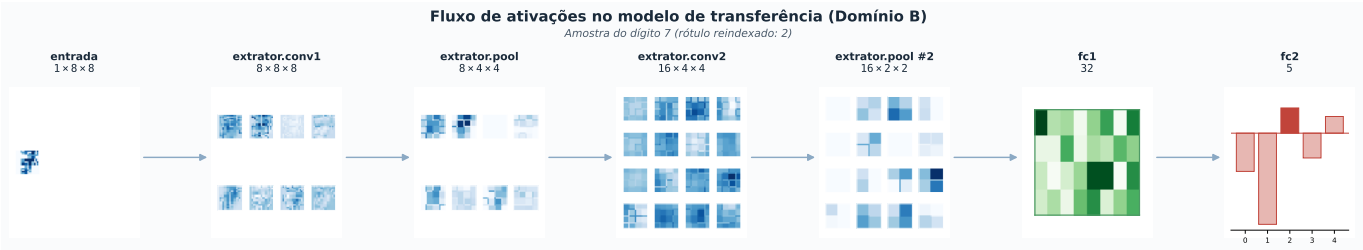


Figure 9.22: Flow of activations and dimensional transformations of the tensors in the transfer learning model when processing a test sample from Domain B (digit 7, reindexed to class 2).

```
Layers captured in the transfer model:
 ['extrator.conv1', 'extrator.pool', 'extrator.conv2', 'extrator.pool #2', 'fc1', 'fc2']
```

Analysis of Experiment 1

The model trained from scratch achieves superior accuracy compared to the model with transfer learning and a frozen feature extractor. This result characterizes a case of **negative transfer** and stems from three factors:

1. **Low capacity:** The feature extractor has only 16 filters of size 3×3, insufficient to learn generalizable representations.
2. **Domain specialization:** Training on digits 0 through 4 produces filters that are poorly discriminative for digits 5 through 9.
3. **Lack of adaptation:** Freezing prevents the feature extractor from adjusting its filters to the new task.

Pedagogical Prompt

This experiment uses a small feature extractor trained on a restricted domain. Would the result be different if the feature extractor had learned its representations on a dataset with millions of images and great object diversity?

Experiment 2 — When Transfer Actually Works (*Pre-trained ResNet-18*)

The second experiment repeats the same structure as the first — few training examples, two classes, comparison between strategies — but replaces the handcrafted extractor of 16 filters with the ***ResNet-18***, an 18-layer architecture pre-trained on *ImageNet* (1.4 million images, $1,000$ categories), and the synthetic digit dataset with real photographs from the **Oxford-IIIT Pet Dataset** (PARKHI, 2012).

The task: distinguish two canine breeds — **Pug** and **Boxer** — from only 15 training photographs per class.

Why this scenario?

The challenge here is not the visual similarity between the breeds — Pug and Boxer have quite distinct sizes and proportions — but rather the scarcity of data: only 30 real photographs in total, with no synthetic images whatsoever. It is the type of low-data-budget problem that, in practice, motivates the use of pre-trained networks: there is neither time nor resources to photograph and label thousands of dogs before training a classifier from scratch.

Block 1: Loading the Real *Dataset* and Sparse Sampling

1. **Source:** the *Oxford-IIIT Pet Dataset* (PARKHI, 2012) is loaded via `torchvision.datasets.OxfordIIITPet`, which automatically downloads the $7,349$ photographs and their breed labels on the first run.
2. **Filtering:** only the two breeds of interest (`Pug`, `Boxer`) are kept.

3. **Deliberate sparsity:** only 15 training photographs per class (30 in total) are sampled — the remainder constitutes the test set, used exclusively for evaluation.

Figure 9.23 displays training samples for each breed.

```
import random

RACAS_ALVO = ["Pug", "Boxer"]
N_TREINO_POR_CLASSE = 15
N_TESTE_POR_CLASSE = 20

# 1. Download of the full dataset (37 breeds) - CC BY-SA 4.0 license
pets_completo = OxfordIIITPet(
    root="dados_pets", split="trainval", target_types="category", download=True
)
nomes_racas = pets_completo.classes
indices_alvo = [nomes_racas.index(r) for r in RACAS_ALVO]

# 2. Filtering of the two breeds of interest, separated by class
por_classe = {idx: [] for idx in indices_alvo}
for img, lbl in pets_completo:
    if lbl in indices_alvo:
        por_classe[lbl].append(img)

# 3. Sampling: few training images, more test images
rng = random.Random(42)
imgs_treino, y_treino, imgs_teste, y_teste = [], [], [], []
for classe_idx, idx_original in enumerate(indices_alvo):
    imgs_raca = por_classe[idx_original][:]
    rng.shuffle(imgs_raca)
    imgs_treino += imgs_raca[:N_TREINO_POR_CLASSE]
    y_treino += [classe_idx] * N_TREINO_POR_CLASSE
    imgs_teste += imgs_raca[N_TREINO_POR_CLASSE : N_TREINO_POR_CLASSE + N_TESTE_POR_CLASSE]
    y_teste += [classe_idx] * N_TESTE_POR_CLASSE

print(f"Training: {len(imgs_treino)} images | Test: {len(imgs_teste)} images")

amostras_pil = imgs_treino[:4] + imgs_treino[N_TREINO_POR_CLASSE:N_TREINO_POR_CLASSE + 4]
amostras_exibicao = [np.array(img.convert("RGB")) for img in amostras_pil]  # PIL -> ndarray
titulos_exibicao = [RACAS_ALVO[0]] * 4 + [RACAS_ALVO[1]] * 4
mm.show(amostras_exibicao, titles=titulos_exibicao, cols=4, figsize=(11, 6))
```

Figure 9.23

Block 2: Three Strategies on the Same Architecture

To isolate the effect of transfer learning, the three strategies reuse **exactly the same architecture** (*ResNet-18*), differing only in the origin of the weights and which parameters remain trainable:

1. **`from_scratch`:** random weights (`weights=None`) — equivalent to training the *ResNet-18* architecture entirely from scratch, as in Block 2 of Experiment 1.
2. **`frozen`:** weights pre-trained on *ImageNet*, with `requires_grad = False` in all convolutional layers — only the new final layer is trained.
3. **`fine_tuning`:** weights pre-trained on *ImageNet* as a starting point, but **without** freezing — the entire network adapts to the new domain, with a low learning rate to avoid destroying prior knowledge.

In all cases, the final layer `fc` is replaced by `nn.Linear(fc.in_features, 2)`, corresponding to the two target breeds.

```python
transformacao_resnet = T.Compose([
    T.Resize((224, 224)),
    T.ToTensor(),
    T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

def prepara_tensores(imgs, labels):
    X = torch.stack([transformacao_resnet(img.convert("RGB")) for img in imgs])
    y = torch.tensor(labels, dtype=torch.long)
    return X, y

X_tr, y_tr = prepara_tensores(imgs_treino, y_treino)
X_te, y_te = prepara_tensores(imgs_teste, y_teste)

def cria_modelo_pets(estrategia):
    pesos = None if estrategia == "do_zero" else models.ResNet18_Weights.DEFAULT
    modelo = models.resnet18(weights=pesos)
    if estrategia == "congelado":
        for p in modelo.parameters():
            p.requires_grad = False
    modelo.fc = nn.Linear(modelo.fc.in_features, len(RACAS_ALVO))
    return modelo

modelo_do_zero = cria_modelo_pets("do_zero")
modelo_congelado = cria_modelo_pets("congelado")
modelo_fine_tuning = cria_modelo_pets("fine_tuning")
```

Block 3: Comparative Training and Accuracy Analysis

Reusing the generic functions `treinar` and `calcular_acuracia`, defined in Block 2 of Experiment 1, the three models are trained on the same set of 30 photographs and evaluated on the test set (images never seen during training):

- The `do_zero` model tends to **overfit** quickly to the 30 training photographs, without generalizing to the test set — 30 examples are drastically insufficient to adjust the 11 million parameters of the *ResNet-18* from scratch.
- The `congelado` model should already achieve considerably higher accuracy, as it reuses, without any adjustment, generic visual features (edges, textures, contours) learned on *ImageNet* — only the new linear layer needs to be adjusted to the 30 photographs.
- The `fine_tuning` model tends to match or surpass the frozen extractor, as it starts from the same prior knowledge, but also allows fine-tuning of the entire network to the visual particularities of the breeds.

Figure 9.24 summarizes the three results.

Block 4: Qualitative Inspection of Predictions

As in Experiment 1 and in the following section on foliar diagnosis, it is instructive to observe individually some predictions of the best model (typically `fine_tuning` or `frozen`) on real test photographs, comparing the predicted label with the actual breed.

```python
# Explicit cleanup of downloaded data and memory release
if FLAG_LIMPAR_DADOS:
    if os.path.exists('./dados_pets'):
        shutil.rmtree('./dados_pets')
        print('  Temporary data directory ./pets_data removed successfully.')

    if torch.cuda.is_available():
        torch.cuda.empty_cache()
```

```python
torch.manual_seed(42)

configuracoes = [
    ("Do zero",              modelo_do_zero,      None, 2e-3),
    ("Extrator congelado",   modelo_congelado,    "fc", 1e-3),
    ("Fine-tuning completo", modelo_fine_tuning,  None, 1e-4),
]

resultados_pets = {}
for nome, modelo, alvo_params, taxa in configuracoes:
    parametros = modelo.fc.parameters() if alvo_params == "fc" else None
    treinar(modelo, X_tr, y_tr, epocas=15, lr=taxa, tam_lote=8, parametros=parametros)
    resultados_pets[nome] = calcular_acuracia(modelo, X_te, y_te)
    print(f"{nome}: {resultados_pets[nome]*100:.1f}%")
```

```python
plt.figure(figsize=(5.5, 4))
cores = ["#dc2626", "#f59e0b", "#16a34a"]
plt.bar(resultados_pets.keys(), resultados_pets.values(), color=cores)
plt.ylim(0, 1.05)
plt.axhline(0.5, color="gray", linestyle="--", linewidth=1, label="Chute aleatório (50%)")
plt.ylabel("Acurácia no teste")
plt.title("Pug vs. Boxer - 15 fotos de treino/classe")
for i, v in enumerate(resultados_pets.values()):
    plt.text(i, v + 0.02, f"{v*100:.1f}%", ha="center")
plt.xticks(rotation=10)
plt.legend()
plt.tight_layout()
plt.show()
```

Figure 9.24

```python
melhor_modelo = modelo_fine_tuning  # or frozen_model, according to the result of Block 3
melhor_modelo.eval()

idx_amostras = list(range(4)) + list(range(N_TESTE_POR_CLASSE, N_TESTE_POR_CLASSE + 4))

imgs_pred, titulos_pred = [], []
with torch.no_grad():
    for idx in idx_amostras:
        entrada = X_te[idx].unsqueeze(0)
```

```python
        pred_idx = melhor_modelo(entrada).argmax(dim=1).item()
        real_idx = y_te[idx].item()
        marcador = " " if pred_idx == real_idx else " "
        imgs_pred.append(np.array(imgs_teste[idx].convert("RGB")))  # PIL -> ndarray
        titulos_pred.append(f"{marcador} previsto: {RACAS_ALVO[pred_idx]}\n"
                            f"real: {RACAS_ALVO[real_idx]}")

mm.show(imgs_pred, titles=titulos_pred, cols=4, figsize=(12, 7))
```

Figure 9.25

Experiment 3 — Phytosanitary Diagnosis with Transfer Learning

The previous experiment showed that a *ResNet-18* pre-trained on **ImageNet** can adapt to a new task using few samples. Now, the same strategy is applied to a phytosanitary diagnosis problem. The *ResNet-18* must classify images of leaves into three categories: `["folha_saudavel", "folha_doente", "sintoma_desconhecido"]`.

- **`folha_saudavel`:** leaf without visible lesions.
- **`folha_doente`:** leaf with dark spots that simulate a fungal disease.
- **`sintoma_desconhecido`:** leaf with yellowish chlorosis, representing a pattern different from the known disease.

🌱 Why this scenario?

Phytosanitary diagnosis constitutes an important application of CV in precision agriculture. A model pre-trained on **ImageNet** can reuse features such as edges, textures, and color patterns to learn this new task with few images.

Block 1: Generation of the Synthetic *Dataset*

This block generates a synthetic set with **30** images per class for training and **8** for validation, totaling **90** and **24** images, respectively.

1. **Leaf generation:** The `desenha_folha_base` function creates the leaf outline, varying size, orientation, and shade of green.
2. **Disease simulation:** The `aplica_manchas_doenca` function adds irregular dark spots that simulate fungal lesions.
3. **Simulation of another symptom:** The `aplica_sintoma_desconhecido` function adds yellowish regions representing a pattern distinct from the known disease.
4. **Sample visualization:** Figure 9.26 presents examples of the three categories of the synthetic set.

```
CLASSES_FOLHA = ["folha_saudavel", "folha_doente", "sintoma_desconhecido"]


def desenha_folha_base(tam_img, rng):
    '''Draws the oval outline of a green leaf with a central vein,
    with small variations in tone, size and orientation between samples.'''
    img = np.full((tam_img, tam_img, 3), 245, dtype=np.uint8)  # light background
    cx, cy = tam_img // 2, tam_img // 2
    eixo_a = rng.randint(int(tam_img * 0.30), int(tam_img * 0.38))
    eixo_b = rng.randint(int(tam_img * 0.20), int(tam_img * 0.26))
    angulo = rng.uniform(-15, 15)
    verde = (rng.randint(40, 70), rng.randint(120, 160), rng.randint(40, 70))
    cv2.ellipse(img, (cx, cy), (eixo_a, eixo_b), angulo, 0, 360, verde, -1, cv2.LINE_AA)
    ang_rad = np.deg2rad(angulo)
    dx, dy = np.cos(ang_rad), np.sin(ang_rad)
    p1 = (int(cx - eixo_a * dx), int(cy - eixo_a * dy))
    p2 = (int(cx + eixo_a * dx), int(cy + eixo_a * dy))
    cv2.line(img, p1, p2, (25, 90, 25), 2, cv2.LINE_AA)  # central vein
    return img, (cx, cy, eixo_a, eixo_b, angulo)


def aplica_manchas_doenca(img, centro_folha, rng, n_manchas=(4, 8)):
    '''Simulates leaf lesions: dark spots with irregular edges
    (typical pattern of fungal diseases).'''
    cx, cy, eixo_a, eixo_b, _ = centro_folha
    for _ in range(rng.randint(*n_manchas)):
        raio = rng.randint(1, 3)
```

```
        px = cx + rng.randint(-int(eixo_a * 0.7), int(eixo_a * 0.7))
        py = cy + rng.randint(-int(eixo_b * 0.7), int(eixo_b * 0.7))
        cor_mancha = (rng.randint(50, 90), rng.randint(25, 45), rng.randint(10, 25))
        cv2.circle(img, (px, py), raio, cor_mancha, -1, cv2.LINE_AA)
        cv2.circle(img, (px, py), raio + 1, (120, 85, 30), 1, cv2.LINE_AA)  # halo
    return img


def aplica_sintoma_desconhecido(img, centro_folha, rng):
    '''Simulates a distinct pattern (yellowish mottling/chlorosis), different
    from the dark spots of the known disease.'''
    cx, cy, eixo_a, eixo_b, _ = centro_folha
    for _ in range(rng.randint(3, 5)):
        eixo_m = (rng.randint(1, 3), rng.randint(2, 3))
        px = cx + rng.randint(-int(eixo_a * 0.6), int(eixo_a * 0.6))
        py = cy + rng.randint(-int(eixo_b * 0.6), int(eixo_b * 0.6))
        cor_clorose = (rng.randint(200, 235), rng.randint(195, 225), rng.randint(50, 90))
        ang_m = rng.uniform(0, 180)
        cv2.ellipse(img, (px, py), eixo_m, ang_m, 0, 360, cor_clorose, -1, cv2.LINE_AA)
    return img


def aplica_ruido_sal_pimenta(img, prop_ruido=0.02, rng=None):
    '''Applies random salt noise (white dots) and pepper noise (black dots).
    prop_noise: fraction of altered pixels (e.g., 0.02 = 2% of pixels).'''
    if prop_ruido <= 0:
        return img

    img_ruido = img.copy()
    num_pixels = int(prop_ruido * img.shape[0] * img.shape[1])
    n_sal = num_pixels // 2
    n_pimenta = num_pixels - n_sal

    # Applies Salt (White - [255, 255, 255])
    for _ in range(n_sal):
        y = rng.randint(0, img.shape[0] - 1)
        x = rng.randint(0, img.shape[1] - 1)
        img_ruido[y, x] = [255, 255, 255]

    # Applies Pepper (Black - [0, 0, 0])
    for _ in range(n_pimenta):
        y = rng.randint(0, img.shape[0] - 1)
        x = rng.randint(0, img.shape[1] - 1)
        img_ruido[y, x] = [0, 0, 0]

    return img_ruido


def gera_folha(classe_idx, tam_img=128, rng=None, prop_ruido=0.02):
    rng = rng or random.Random()
    img, geometria = desenha_folha_base(tam_img, rng)
    nome = CLASSES_FOLHA[classe_idx]

    if nome == "folha_doente":
        img = aplica_manchas_doenca(img, geometria, rng)
    elif nome == "sintoma_desconhecido":
        img = aplica_sintoma_desconhecido(img, geometria, rng)

    ruido_exp = rng.randint(-3, 3)  # slight exposure variation
    img = np.clip(img.astype(np.int16) + ruido_exp, 0, 255).astype(np.uint8)
```

```
    # Application of Salt and Pepper noise
    img = aplica_ruido_sal_pimenta(img, prop_ruido=prop_ruido, rng=rng)

    return img


def gera_conjunto(n_por_classe, tam_img=128, seed=0, prop_ruido=0.02):
    rng = random.Random(seed)
    imgs, labels = [], []
    for classe_idx in range(len(CLASSES_FOLHA)):
        for _ in range(n_por_classe):
          imgs.append(gera_folha(classe_idx, tam_img=tam_img, rng=rng, prop_ruido=prop_ruido))
          labels.append(classe_idx)
    return imgs, labels


N_POR_CLASSE_TREINO, N_POR_CLASSE_VAL = 30, 8

imgs_treino, labels_treino = gera_conjunto(n_por_classe=N_POR_CLASSE_TREINO, seed=42,
                                           prop_ruido=0.02)
imgs_val, labels_val = gera_conjunto(n_por_classe=N_POR_CLASSE_VAL, seed=123, prop_ruido=0.02)

print(f"Training: {len(imgs_treino)} images ({N_POR_CLASSE_TREINO} per class) | "
      f"Validation: {len(imgs_val)} images ({N_POR_CLASSE_VAL} per class)")

# Display of 2 samples of each class (6 images in total)
amostras_exibir, titulos_exibir = [], []
for classe_idx, nome in enumerate(CLASSES_FOLHA):
    for k in range(2):
        idx = classe_idx * N_POR_CLASSE_TREINO + k
        amostras_exibir.append(imgs_treino[idx])
        titulos_exibir.append(nome)

mm.show(amostras_exibir, titles=titulos_exibir, cols=3, figsize=(10, 7))
```

```
Training: 90 images (30 per class) | Validation: 24 images (8 per class)
```

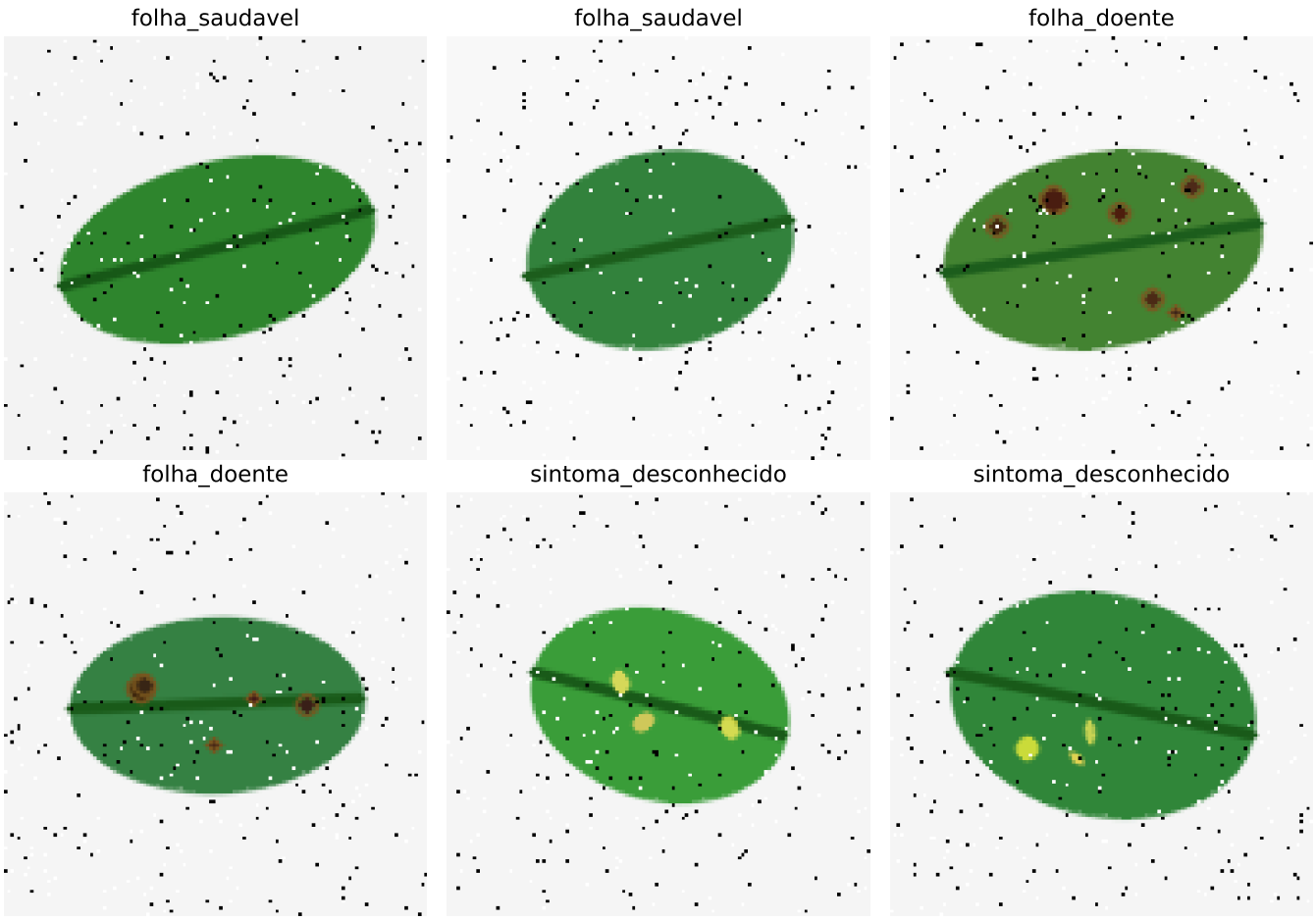


Figure 9.26: Synthetic samples from the leaf diagnosis *dataset*: healthy leaf, diseased leaf (dark spots) and unknown symptom (yellowish chlorosis) with salt-and-pepper noise.

> **Common Pitfall**
>
> Transfer learning requires that each class exhibits distinct visual patterns. Repeating the same image with different labels prevents the classifier layer from learning a decision boundary, since the feature extractor produces nearly identical features for all samples.
> In this experiment, each image is independently generated, with visual patterns consistent with its class (healthy leaf, fungal lesions, or chlorosis), providing sufficient information for training the classifier layer.

Block 2: Preparation of Tensors and Architecture Adaptation

Models such as *ResNet-18* require color images of 224×224 *pixels* normalized according to *ImageNet* statistics ($\mu = [0.485; 0.456; 0.406]$ and $\sigma = [0.229; 0.224; 0.225]$).

1. **Input Transformation (`transforms.Compose`):** resizing and standard normalization are applied to each image of the synthetic *dataset*, producing the tensors `X_treino`/`X_val` and the labels `y_treino`/`y_val` — each example is a genuinely distinct image, associated with the correct label of its class.
2. **Freezing the Extractor:** the loop `for p in modelo_resnet.parameters(): p.requires_grad = False` disables gradients in the pre-trained convolutional layers.
3. **New Final Layer:** the layer `modelo_resnet.fc` is replaced by a new instance `nn.Linear(modelo_resnet.fc.in`
`n_classes_destino)`, freshly initialized and with gradients active by default. The number of input features is obtained dynamically from the original layer itself (`in_features`, equal to 512 in ResNet-18),

rather than being manually fixed in the code — a recommended practice, as it makes the snippet reusable for other variants of the architecture without modifications.

```
# 1. Pipeline of transformations expected by ResNet
transformacao_resnet = T.Compose([
    T.Resize((224, 224)),
    T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])


def prepara_tensores(imgs, labels):
    tensores = [transformacao_resnet(T.functional.to_tensor(img)) for img in imgs]
    X = torch.stack(tensores)
    y = torch.tensor(labels, dtype=torch.long)
    return X, y


# 2. Conversion of the synthetic dataset (Block 1) into normalized tensors
X_treino, y_treino = prepara_tensores(imgs_treino, labels_treino)
X_val, y_val = prepara_tensores(imgs_val, labels_val)

loader_treino = DataLoader(TensorDataset(X_treino, y_treino), batch_size=16, shuffle=True)
loader_val = DataLoader(TensorDataset(X_val, y_val), batch_size=16, shuffle=False)

# 3. Loading of the pre-trained ResNet-18 and freezing of the feature extractor
n_classes_destino = len(CLASSES_FOLHA)
modelo_resnet = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)

for parametro in modelo_resnet.parameters():
    parametro.requires_grad = False

# 4. Replacement of the final layer for the 3 new target classes
modelo_resnet.fc = nn.Linear(modelo_resnet.fc.in_features, n_classes_destino)

print(f"New final layer: {modelo_resnet.fc}")
```

Block 3: Training Loop and Evaluation

With the extractor frozen and the new output layer properly attached, the fine-tuning of the new classification head is carried out.

1. **Focused Optimization:** the *Adam* optimizer receives strictly `modelo_resnet.fc.parameters()`, updating only the new output layer — the rest of the network remains frozen, as defined in Block 2.
2. **Loop Execution:** at each epoch, the model iterates over the training batches, computes the Cross-Entropy loss, and adjusts the weights of the final layer; subsequently, the accuracy is evaluated on the validation set (images never seen during training).
3. **Training Curves:** Figure 9.27 tracks the evolution of the training loss and validation accuracy over the epochs — since the three classes are visually distinct from one another, genuine convergence is expected, well above the 33% threshold corresponding to a random guess among 3 classes.

Block 4: Qualitative Inspection of Predictions

In addition to the aggregated accuracy curve, it is instructive to individually examine some of the model's predictions on the validation set, comparing the predicted label with the true label. Figure 9.28 displays two samples from each class.

```python
dispositivo = torch.device("cuda" if torch.cuda.is_available() else "cpu")
modelo_resnet = modelo_resnet.to(dispositivo)

criterio = nn.CrossEntropyLoss()
otimizador = optim.Adam(modelo_resnet.fc.parameters(), lr=1e-3)

historico_perda, historico_acc = [], []
epocas = 10

for epoca in range(epocas):

    modelo_resnet.train()
    perda_acumulada, n_batches = 0.0, 0

    for X_batch, y_batch in loader_treino:
        X_batch, y_batch = X_batch.to(dispositivo), y_batch.to(dispositivo)

        otimizador.zero_grad()
        saidas = modelo_resnet(X_batch)
        perda = criterio(saidas, y_batch)
        perda.backward()

        otimizador.step()

        perda_acumulada += perda.item()
        n_batches += 1
    historico_perda.append(perda_acumulada / n_batches)

    modelo_resnet.eval()
    acertos, total = 0, 0
    with torch.no_grad():
        for X_batch, y_batch in loader_val:

            X_batch, y_batch = X_batch.to(dispositivo), y_batch.to(dispositivo)
            predicoes = modelo_resnet(X_batch).argmax(dim=1)
            acertos += (predicoes == y_batch).sum().item()
            total += y_batch.size(0)
    acc = acertos / total
    historico_acc.append(acc)

    print(f"Epoch {epoca+1}/{epocas} - loss: {historico_perda[-1]:.4f} - ",
          f" val_accuracy: {acc*100:.1f}%")
```

```python
f=mm.showTrainCurves(
    historico_perda, historico_acc,
    titulo="Fine-Tuning da ResNet-18 - Diagnóstico Foliar",
    subtitulo="Apenas a nova camada linear (fc) é treinada; o extrator permanece congelado",
)
```

Figure 9.27

```
modelo_resnet.eval()

# Two samples from each class in the validation set
idx_amostras = [0, N_POR_CLASSE_VAL, 2 * N_POR_CLASSE_VAL,

                1, N_POR_CLASSE_VAL + 1, 2 * N_POR_CLASSE_VAL + 1]

imgs_pred, titulos_pred = [], []
with torch.no_grad():

    for idx in idx_amostras:
        entrada = X_val[idx].unsqueeze(0).to(dispositivo)
        pred_idx = modelo_resnet(entrada).argmax(dim=1).item()
        real_idx = y_val[idx].item()

        marcador = " " if pred_idx == real_idx else " "
        imgs_pred.append(imgs_val[idx])
        titulos_pred.append(f"{marcador} previsto: {CLASSES_FOLHA[pred_idx]}\n"+
                            f"real: {CLASSES_FOLHA[real_idx]}")


mm.show(imgs_pred, titles=titulos_pred, cols=3, figsize=(10, 7))
```

Figure 9.28

Analysis of Experiment 3

The *ResNet-18* achieves high accuracy even when using a reduced set of synthetic images. This result shows that the representations learned on **ImageNet** remain useful in a completely different domain, requiring only the adaptation of the classifier layer.

The experiment also illustrates a common situation in real-world applications, where the availability of labeled data is limited. In such scenarios, transfer learning reduces training time and allows obtaining models with good performance even without training the entire network.

🧠 Comparative Synthesis — When Does Transfer Learning Work?

The three experiments show that transfer learning depends on the generalization capacity of the feature extractor.

- **Experiment 1:** a small extractor, trained on a restricted domain, learns poorly generalizable representations and can produce **negative transfer**.
- **Experiment 2:** a *ResNet-18* pre-trained on **ImageNet** transfers general representations to a dog and cat breed classification task, achieving high accuracy with few samples.
- **Experiment 3:** the same strategy adapts the model to a plant health diagnosis problem, showing that a single extractor can serve as a foundation for different application domains.

Comparison of Transfer Approaches

Table 9.2 summarizes the results obtained in the three experiments.

Table 9.2: Comparison among the three transfer learning scenarios presented in this section.

Aspect	Experiment 1	Experiment 2	Experiment 3
Feature extractor	Small CNN	*ResNet-18*	*ResNet-18*
Extractor training	Digits (0–4)	**ImageNet**	**ImageNet**
Generalization capacity	Low	High	High

Aspect	Experiment 1	Experiment 2	Experiment 3
New task	Digits (5–9)	Dog and cat breeds	Plant health diagnosis
Result	Negative transfer	Positive transfer	Positive transfer

9.5.2 Object Detection

The previous experiments showed how transfer learning adapts pre-trained models for image classification tasks. The same principle also underpins object detection architectures, in which a pre-trained feature extractor provides general visual representations, while specialized modules locate and classify objects in the image.

The following sections present **Faster R-CNN** as an example of a pre-trained detector used directly for inference, and then a complete fine-tuning experiment with the **YOLO** architecture.

9.5.2.1 Faster R-CNN: Pretrained Detector

Object detection extends the use of pretrained models to a task more complex than classification. **Faster R-CNN** uses a pretrained CNN, such as *ResNet-50*, as a **feature extractor** (*backbone*) and adds specialized modules to locate and classify objects.

Regarding this extractor, the architecture incorporates two main "heads":

- ***Region Proposal Network* (RPN):** proposes regions of the image with a high probability of containing objects.
- **Classifier head:** refines these regions, assigns a class to each object, and adjusts its bounding boxes.

The following code uses a Faster R-CNN with weights pretrained on **COCO** to detect objects in an image, producing their classes, coordinates, and confidence scores.

> 🔍 Where is transfer learning here?
>
> Unlike the previous experiments, this example **does not perform fine-tuning**. The model only performs inference (`eval()`), directly reusing the weights of the *backbone*, the RPN, and the classifier head trained on **COCO**.
> Adaptation to a new domain would require replacing the `box_predictor` layer with a new classification head, compatible with the classes of the application, and training it on a set of annotated images. This procedure follows the same principle presented in the transfer learning section and constitutes the usual workflow for specific applications, such as pest detection, manufacturing defects, or vehicles.

1. **COCO categories:** The code retrieves the class names from the metadata of the weights (`FasterRCNN_ResNet50_FPN_Weights.DEFAULT.meta["categories"]`). Although this list contains 91 entries for historical reasons of the COCO annotation format, only 80 correspond to object categories.

2. **Inference:** The image loaded by `mm.read()` is converted into a *tensor* and processed by the model in evaluation mode (`eval()`). The code keeps only the detections with confidence greater than 80%.

3. **Image annotation:** For each detected object, the code draws the bounding box (`cv2.rectangle`) and writes the predicted class and its confidence (`cv2.putText`).

4. **Visualization:** Figure 9.29 presents the annotated image with the detections made by the model.

9.5.2.2 Object Detection and Transfer Learning with YOLO

The previous sections applied transfer learning to **image classification** problems, where the model associates a single label with the entire image. In this section, the same principle is extended to **object detection**, a task that requires simultaneously identifying **what** is present in the image and **where** each object is located.

```python
# 1. Loading the image and the COCO category names
url_imagem = "https://raw.githubusercontent.com/pytorch/hub/master/images/dog.jpg"
url_imagem = "http://images.cocodataset.org/val2017/000000039769.jpg"
img = mm.read(url_imagem)

pesos_coco = FasterRCNN_ResNet50_FPN_Weights.DEFAULT

categorias_coco = pesos_coco.meta["categories"]  # Mapping index -> class name

# 2. Loading the pre-trained Faster R-CNN model
modelo_detection = fasterrcnn_resnet50_fpn(weights=pesos_coco).eval()

# 3. Running inference without gradient calculation

with torch.no_grad():
    predicao = modelo_detection([to_tensor(img)])[0]

# 4. Filtering detections with confidence higher than 80%
limiar_confianca = 0.8
mascara_confianca = predicao["scores"] >= limiar_confianca


caixas_filtradas = predicao["boxes"][mascara_confianca].numpy()
scores_filtrados = predicao["scores"][mascara_confianca].numpy()
labels_filtrados = predicao["labels"][mascara_confianca].numpy()

img_com_caixas = img.copy()


# 5. Drawing the bounding boxes and class labels
for box, score, label_idx in zip(caixas_filtradas, scores_filtrados, labels_filtrados):
    x1, y1, x2, y2 = box.astype(int)
    nome_classe = categorias_coco[label_idx]
    texto_rotulo = f"{nome_classe}: {score:.2f}"


    # Draws the red rectangle (RGB: 255, 0, 0) with 3 pixel thickness
    cv2.rectangle(img_com_caixas, (x1, y1), (x2, y2), (255, 0, 0), 3)

    # Writes the class and confidence above the bounding box
    cv2.putText(

        img_com_caixas,
        texto_rotulo,
        (x1, max(y1 - 10, 20)),
        cv2.FONT_HERSHEY_SIMPLEX,
        0.8,
        (255, 0, 0),

        2,
        cv2.LINE_AA,
    )

# 6. Graphical display of the resulting image
mm.show(img_com_caixas, title="Faster R-CNN (COCO) - Detection with Class and Confidence")
```

Figure 9.29

The previous subsection presented **Faster R-CNN** as an example of a pretrained detector used directly for inference, without any adaptation to the new domain. In this experiment, the model undergoes a **fine-tuning** step: starting from a **YOLO** (*You Only Look Once*) architecture pretrained on the **COCO** dataset, the network is adapted to detect and classify objects from a new domain.

Unlike Faster R-CNN, which performs detection in two stages, the **YOLO** family adopts a single-stage detector architecture, estimating, in a single pass through the network, the bounding boxes, the confidence of each detection, and the corresponding class. This strategy reduces computational cost and enables real-time applications.

As an example, the experiment uses a synthetic dataset of geometric shapes (triangles, squares, stars, among others), with variations in color, size, rotation, and degradation by salt-and-pepper noise.

Block 1: Generation of the Synthetic *Dataset*

The following block generates a synthetic set for training and evaluating the detector. Each image contains between one and three objects belonging to one of the nine classes:

```
CLASSES = [
    'Triangle', 'Square', 'Pentagon', 'Hexagon',
    'Heptagon', 'Circle', 'Ellipse', 'Star', 'Cross'
]
```

1. **Shape generation:** The functions `poligono_regular`, `poligono_estrela`, and `poligono_cruz` construct the coordinates of the objects. The function `desenha_objeto` draws each shape with random position, size, orientation, and color and calculates its *bounding box*.
2. **Annotation in YOLO format:** The function `gera_imagem_ruidosa` generates between one and three objects per image and converts each *bounding box* to the YOLO format, represented by the class and the normalized coordinates of the center, width, and height.
3. **Image degradation:** The function `adiciona_ruido_sal_pimenta` adds impulsive noise, simulating acquisition imperfections.
4. **Sample visualization:** Figure 9.30 presents examples of the synthetic set with the *bounding boxes* overlaid by the function `mm.showBoundBox()`.

```
CLASSES = [
    'Triangle', 'Square', 'Pentagon', 'Hexagon',
    'Heptagon', 'Circle', 'Ellipse', 'Star', 'Cross'
]
N_LADOS = {'Triangle': 3, 'Square': 4, 'Pentagon': 5, 'Hexagon': 6, 'Heptagon': 7}

def poligono_regular(cx, cy, r, n_lados, rot_graus):
    ang0 = np.deg2rad(rot_graus - 90)
    angs = ang0 + 2 * np.pi * np.arange(n_lados) / n_lados
    return np.stack([cx + r * np.cos(angs), cy + r * np.sin(angs)], axis=1)

def poligono_estrela(cx, cy, r_externo, rot_graus, n_pontas=5):
    r_interno = r_externo * 0.45
    ang0 = np.deg2rad(rot_graus - 90)
    angs = ang0 + np.pi * np.arange(2 * n_pontas) / n_pontas
    raios = np.where(np.arange(2 * n_pontas) % 2 == 0, r_externo, r_interno)
    return np.stack([cx + raios * np.cos(angs), cy + raios * np.sin(angs)], axis=1)

def poligono_cruz(cx, cy, r, rot_graus, espessura_rel=0.35):
    w = r * espessura_rel
    base = np.array([
        (-w, -r), (w, -r), (w, -w), (r, -w), (r, w), (w, w),
        (w, r), (-w, r), (-w, w), (-r, w), (-r, -w), (-w, -w),
```

```
    ])
    theta = np.deg2rad(rot_graus)
    R = np.array([[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.cos(theta)]])
    return base @ R.T + np.array([cx, cy])

def desenha_objeto(img, classe_idx, cx, cy, tamanho, rotacao, cor):
    nome = CLASSES[classe_idx]
    if nome in N_LADOS:
        pts = poligono_regular(cx, cy, tamanho, N_LADOS[nome], rotacao)
        cv2.fillPoly(img, [pts.astype(np.int32)], cor)
        xs, ys = pts[:, 0], pts[:, 1]
    elif nome == 'Star':
        pts = poligono_estrela(cx, cy, tamanho, rotacao)
        cv2.fillPoly(img, [pts.astype(np.int32)], cor)
        xs, ys = pts[:, 0], pts[:, 1]
    elif nome == 'Cross':
        pts = poligono_cruz(cx, cy, tamanho, rotacao)
        cv2.fillPoly(img, [pts.astype(np.int32)], cor)
        xs, ys = pts[:, 0], pts[:, 1]
    elif nome == 'Circle':
        cv2.circle(img, (int(cx), int(cy)), int(tamanho), cor, -1)
        xs, ys = np.array([cx - tamanho, cx + tamanho]), np.array([cy - tamanho, cy +
tamanho])
    else:  # Ellipse
        eixo = (int(tamanho), int(tamanho * 0.6))
        cv2.ellipse(img, (int(cx), int(cy)), eixo, rotacao, 0, 360, cor, -1)
        ang = np.deg2rad(rotacao)
        dx = np.hypot(eixo[0] * np.cos(ang), eixo[1] * np.sin(ang))
        dy = np.hypot(eixo[0] * np.sin(ang), eixo[1] * np.cos(ang))
        xs, ys = np.array([cx - dx, cx + dx]), np.array([cy - dy, cy + dy])
    return xs.min(), ys.min(), xs.max(), ys.max()

def adiciona_ruido_sal_pimenta(img, quantidade=0.05):
    img_ruidosa = img.copy()
    h, w, c = img_ruidosa.shape
    num_ruido = int(quantidade * h * w)

    # White (255, 255, 255)
    coords_sal = [np.random.randint(0, i - 1, num_ruido) for i in (h, w)]
    img_ruidosa[coords_sal[0], coords_sal[1]] = [255, 255, 255]

    # Black (0, 0, 0)
    coords_pimenta = [np.random.randint(0, i - 1, num_ruido) for i in (h, w)]
    img_ruidosa[coords_pimenta[0], coords_pimenta[1]] = [0, 0, 0]

    return img_ruidosa

def gera_imagem_ruidosa(tam_img=160, n_objetos=(1, 3), taxa_ruido=0.01, rng=None):
    rng = rng or random.Random()
    img_limpa = np.full((tam_img, tam_img, 3), 255, dtype=np.uint8)
    anotacoes = []

    for _ in range(rng.randint(*n_objetos)):
        classe_idx = rng.randrange(len(CLASSES))
        tamanho = rng.randint(tam_img // 10, tam_img // 5)
        cx = rng.randint(tamanho + 2, tam_img - tamanho - 2)
        cy = rng.randint(tamanho + 2, tam_img - tamanho - 2)
        rotacao = rng.uniform(0, 360)
        cor = tuple(rng.sample(range(30, 226), 3))

        x0, y0, x1, y1 = desenha_objeto(img_limpa, classe_idx, cx, cy, tamanho, rotacao, cor)
```

```
        x0, y0 = max(x0, 0), max(y0, 0)
        x1, y1 = min(x1, tam_img), min(y1, tam_img)

        # YOLO format: (class, x_center, y_center, width, height) normalized
        xc, yc = (x0 + x1) / 2 / tam_img, (y0 + y1) / 2 / tam_img
        w, h = (x1 - x0) / tam_img, (y1 - y0) / tam_img
        anotacoes.append((classe_idx, xc, yc, w, h))

    img_ruidosa = adiciona_ruido_sal_pimenta(img_limpa, quantidade=taxa_ruido)
    return img_ruidosa, anotacoes
```

```
# Generation of 5 samples for initial display at the top of the project
n_amostras_iniciais = 5
rng_demo = random.Random(42)

imgs_demo = []
titulos_demo = []

for idx in range(n_amostras_iniciais):
    img_ruid, anotacoes = gera_imagem_ruidosa(rng=rng_demo)

    # Temporary recording of the annotation for native reading by mm.showBoundBox
    filename_temp = f"temp_label_{idx}.txt"
    with open(filename_temp, "w") as f:
        for c, xc, yc, w, h in anotacoes:
            f.write(f"{c} {xc:.4f} {yc:.4f} {w:.4f} {h:.4f}\n")

    img_anotada = mm.showBoundBox(img_ruid, filename=filename_temp, fmt="yolo", show=False)
    imgs_demo.append(img_anotada)
    titulos_demo.append(f"Amostra {idx+1}")

# Display of the panel of 5 samples
mm.show(
    imgs_demo,
    titles=titulos_demo,
    cols=n_amostras_iniciais,
    figsize=(14, 3)
)
```

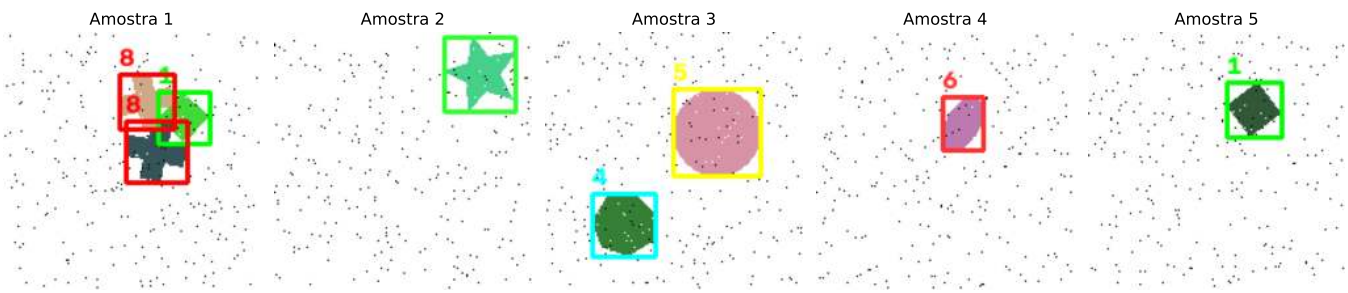


Figure 9.30: Initial samples of the noisy synthetic geometric objects *dataset* with YOLO-format bounding boxes overlaid.

Block 2: Dataset Organization and Creation of the `data.yaml` File

This block organizes the dataset in the format expected by the **Ultralytics YOLO** library. The images and annotations are distributed into separate directories for training and validation, while the `data.yaml` file gathers the necessary information for training the detector.

```
shapes_dataset/
  data.yaml
  images/
     train/
     val/
  labels/
      train/
      val/
```

1. **Dataset generation:** The code creates **90** images for training and **20** for validation. For each image, it writes a `.txt` file containing one line per object, in YOLO format (`class`, `x_c`, `y_c`, `width`, `height`), with all coordinates normalized.

2. **File organization:** The images are stored in `images/train` and `images/val`, while the corresponding annotations are written to `labels/train` and `labels/val`, preserving the same file name.

3. **Creation of the `data.yaml` file:** The code automatically generates the configuration file containing the dataset path, the training and validation directories, and the mapping between numeric indices and the names of the nine classes.

```python
base_dir = "shapes_dataset"
rng_global = random.Random(42)

for split, n_imgs in [("train", 90), ("val", 20)]:
    os.makedirs(f"{base_dir}/images/{split}", exist_ok=True)
    os.makedirs(f"{base_dir}/labels/{split}", exist_ok=True)

    for i in range(n_imgs):
        img_ruid, anotacoes = gera_imagem_ruidosa(rng=rng_global)
        cv2.imwrite(f"{base_dir}/images/{split}/{i:04d}.jpg", img_ruid)

        with open(f"{base_dir}/labels/{split}/{i:04d}.txt", "w") as f:
            for c, xc, yc, w, h in anotacoes:
                f.write(f"{c} {xc:.4f} {yc:.4f} {w:.4f} {h:.4f}\n")

with open(f"{base_dir}/data.yaml", "w") as f:
    f.write(
        f"path: {os.path.abspath(base_dir)}\n"
        "train: images/train\nval: images/val\nnnames:\n"
    )
    for i, nome in enumerate(CLASSES):
        f.write(f"  {i}: {nome}\n")

print("Noisy dataset generated successfully: 90 training images and 20 validation images.")
```

```
Noisy dataset generated successfully: 90 training images and 20 validation images.
```

Block 3: Pre-processing with Median Filter

This block applies pre-processing to reduce the effect of salt-and-pepper noise introduced during *dataset* generation. The **Median Filter** (`cv2.medianBlur`) removes this type of degradation while better preserving object edges than conventional smoothing filters.

1. **Image filtering:** The code applies a median filter with a 3×3 window to the noisy image, reducing impulsive *pixels* without altering the dataset annotations.

2. **Comparative visualization:** Figure 9.31 compares the original image and the filtered image, keeping the *bounding boxes* overlaid using the `mm.showBoundBox()` function.

```
# 1. Loading the first noisy sample from the dataset
caminho_img = f"{base_dir}/images/train/0000.jpg"
caminho_label = f"{base_dir}/labels/train/0000.txt"

img_ruidosa = mm.read(caminho_img)

# 2. Pre-processing with Median Filter (3x3 window)
img_filtrada = cv2.medianBlur(img_ruidosa, ksize=3)

# 3. Overlaying bounding boxes with mm.showBoundBox
img_ruid_anotada = mm.showBoundBox(img_ruidosa, filename=caminho_label, fmt="yolo",
show=False)
img_filt_anotada = mm.showBoundBox(img_filtrada, filename=caminho_label, fmt="yolo",
show=False)

# 4. Comparative display with mm.show
mm.show(
    [img_ruid_anotada, img_filt_anotada],
    titles=[
        "1. Original Noisy Image (Salt and Pepper)",
        "2. Pre-processed (Median Filter 3x3)"
    ],
    cols=2,
    figsize=(9, 4)
)
```

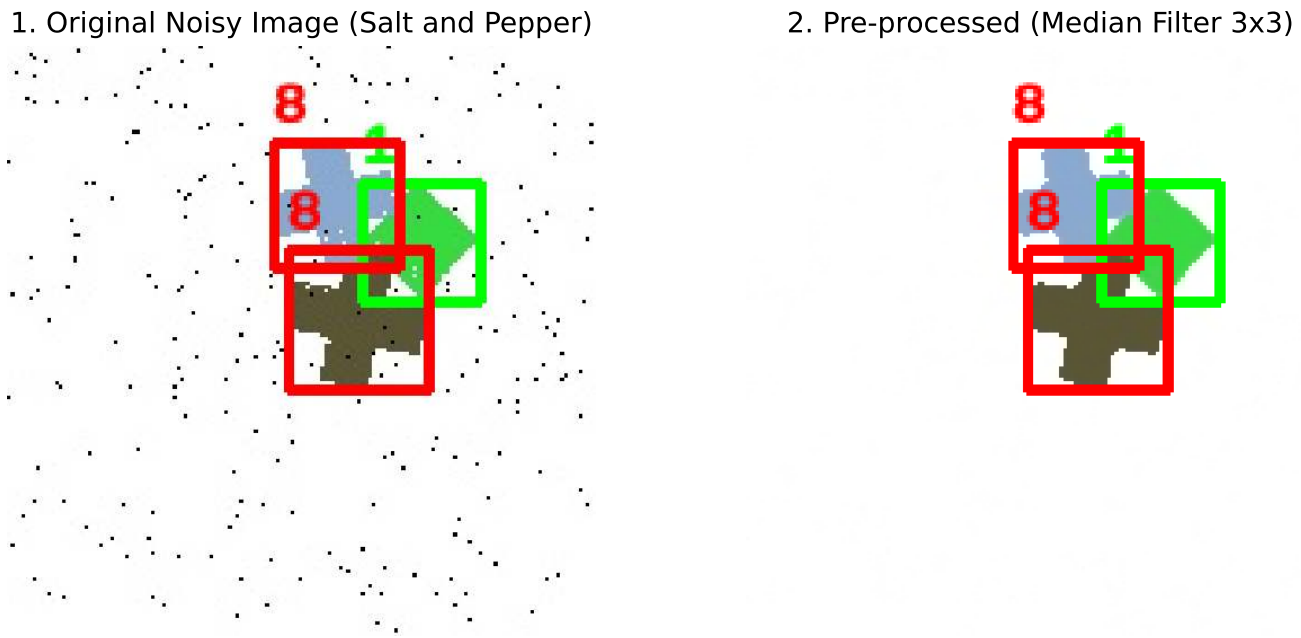


Figure 9.31: Comparison between the original image with salt-and-pepper noise and the image after applying the Median Filter (3x3). The *bounding boxes* in YOLO format remain unchanged.

Block 4: Dataset Preprocessing and Fine-Tuning of YOLOv8

This block applies the Median Filter to the image set and performs fine-tuning of the pre-trained **YOLOv8n** detector on the **COCO** dataset. Filtering reduces the effect of impulsive noise introduced during image generation, while training adapts the network parameters to the new domain of geometric shapes.

1. **Batch filtering:** The code iterates through the `train` and `val` directories and applies `cv2.medianBlur` with a 3×3 window to all images, preserving the original YOLO annotations.

2. **Detector fine-tuning:** The network `YOLO("yolov8n.pt")`, initially trained on COCO, is adapted to the geometric dataset using the `.train()` function. The training uses images with a resolution of 320×320 pixels for 30 epochs.

3. **Model evaluation:** The `.val()` function computes detection metrics on the validation set, including **precision**, **recall**, and **mAP50** (*mean Average Precision* with an IoU threshold equal to 0.5).

```
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

base_dir = "shapes_dataset"
base_dir_filt = "shapes_dataset_filtrado"

# 1. Creates a FILTERED copy of the dataset in a separate folder
#    (the original dataset in base_dir remains noisy, intact)
for split in ["train", "val"]:
    pasta_imgs_orig = f"{base_dir}/images/{split}"
    pasta_labels_orig = f"{base_dir}/labels/{split}"
    pasta_imgs_filt = f"{base_dir_filt}/images/{split}"
    pasta_labels_filt = f"{base_dir_filt}/labels/{split}"

    os.makedirs(pasta_imgs_filt, exist_ok=True)
    os.makedirs(pasta_labels_filt, exist_ok=True)

    for nome_arq in os.listdir(pasta_imgs_orig):
        if nome_arq.endswith(".jpg"):
            img_ruidosa = cv2.imread(f"{pasta_imgs_orig}/{nome_arq}")
            img_filtrada = cv2.medianBlur(img_ruidosa, ksize=3)
            # writes the filtered version to the NEW folder, does not overwrite the original
            cv2.imwrite(f"{pasta_imgs_filt}/{nome_arq}", img_filtrada)

    # copies the labels (they do not change with the filter)
    for nome_arq in os.listdir(pasta_labels_orig):
        shutil.copy(f"{pasta_labels_orig}/{nome_arq}", f"{pasta_labels_filt}/{nome_arq}")

# 2. data.yaml pointing to the FILTERED dataset
with open(f"{base_dir_filt}/data.yaml", "w") as f:
    f.write(
        f"path: {os.path.abspath(base_dir_filt)}\n"
        "train: images/train\nval: images/val\nnames:\n"
    )
    for i, nome in enumerate(CLASSES):
        f.write(f"  {i}: {nome}\n")

print("Pre-processing completed: filtered dataset saved in a separate folder.\n")

# Download pre-trained YOLOv8 model (yolov8n.pt) if not already present
with contextlib.redirect_stdout(io.StringIO()), \
    contextlib.redirect_stderr(io.StringIO()):
    modelo_yolo = YOLO("yolov8n.pt")
print("YOLOv8 model loaded.")
```

```
Pre-processing completed: filtered dataset saved in a separate folder.

YOLOv8 model loaded.
```

```
# 3. Custom callback to print only the epoch currently running
def on_train_epoch_start(trainer):
```

```
    epoch_atual = trainer.epoch + 1
    total_epochs = trainer.epochs
    # Writes directly to the original stdout (bypassing the silencer)
    sys.__stdout__.write(f"  Processando Época {epoch_atual}/{total_epochs}...\n")
    sys.__stdout__.flush()

# Adds the callback to the model
modelo_yolo.add_callback("on_train_epoch_start", on_train_epoch_start)

# 4. Context manager to silence the Ultralytics garbage (C/C++ and Python)
@contextlib.contextmanager
def silenciar_logs():
    logger = logging.getLogger("ultralytics")
    disabled_state = logger.disabled
    logger.disabled = True

    with open(os.devnull, "w") as fnull:
        old_stdout_fd = os.dup(1)
        old_stderr_fd = os.dup(2)
        try:
            os.dup2(fnull.fileno(), 1)
            os.dup2(fnull.fileno(), 2)
            with contextlib.redirect_stdout(fnull), contextlib.redirect_stderr(fnull):
                yield
        finally:
            os.dup2(old_stdout_fd, 1)
            os.dup2(old_stderr_fd, 2)
            os.close(old_stdout_fd)
            os.close(old_stderr_fd)
            logger.disabled = disabled_state

# Training Execution
print("--- Starting YOLOv8 Training ---")
with silenciar_logs():
    resultados_treino = modelo_yolo.train(
        data=f"{base_dir_filt}/data.yaml",   # <-- trains on the filtered dataset
        epochs=30,
        imgsz=320,
        batch=16,
        device=device,
        verbose=False,
        plots=False
    )
    metricas = modelo_yolo.val(verbose=False)

# 5. Final Metrics
precision = metricas.results_dict["metrics/precision(B)"]
recall = metricas.results_dict["metrics/recall(B)"]
map50 = metricas.results_dict["metrics/mAP50(B)"]

print("\n--- Optimized YOLOv8 Model Performance ---")
print(f"Precision: {precision*100:.2f}%")
print(f"Recall: {recall*100:.2f}%")
print(f"mAP at 50% (IoU 0.50): {map50*100:.2f}%")
```

```
--- Starting YOLOv8 Training ---

--- Optimized YOLOv8 Model Performance ---
Precision: 76.77%
Recall: 91.00%
mAP at 50% (IoU 0.50): 87.84%
```

Block 5: Inference Comparison: Noisy Image and Restored Image

After fine-tuning YOLOv8 on the restored dataset, a visual comparison is performed between detection applied directly to an image degraded by salt-and-pepper noise and the same image after the Median Filter.

The goal is to observe how a simple preprocessing step can influence the quality of predictions from a detector already adapted to the new domain.

1. **Inference with YOLO:** The `modelo_yolo.predict()` function performs detection on both versions of the image, using a confidence threshold of 25% (`conf=0.25`).

2. **Visualization of Predictions:** The `plot()` function generates the images annotated with the bounding boxes and labels predicted by the model. Figure 9.32 presents the comparison between the two scenarios.

```
# 1. Loading a original test sample (without the filter saved in batch)
caminho_teste = f"{base_dir}/images/val/0002.jpg"
img_ruidosa_teste = mm.read(caminho_teste)

# 2. Punctual application of the Median Filter (3x3) for comparison
img_filtrada_teste = cv2.medianBlur(img_ruidosa_teste, ksize=3)

# 3. Inference with the trained YOLOv8 model
pred_ruidosa = modelo_yolo.predict(img_ruidosa_teste, conf=0.25, verbose=False)[0]
pred_filtrada = modelo_yolo.predict(img_filtrada_teste, conf=0.25, verbose=False)[0]

# 4. Extraction of the annotated matrices by the YOLO generator (BGR -> RGB conversion)
img_pred_ruid = cv2.cvtColor(pred_ruidosa.plot(), cv2.COLOR_BGR2RGB)
img_pred_filt = cv2.cvtColor(pred_filtrada.plot(), cv2.COLOR_BGR2RGB)

# 5. Standardized comparative display via mm.show
mm.show(
    [img_pred_ruid, img_pred_filt],
    titles=[
        f"Inference on the Noisy Image ({len(pred_ruidosa.boxes)} objects)",
        f"Inference on the Filtered Image ({len(pred_filtrada.boxes)} objects)"
    ],
    cols=2,
    figsize=(10, 4)
)
```

Figure 9.32

```
# Explicit cleaning of downloaded data and memory release
if FLAG_LIMPAR_DADOS:
    if os.path.exists(base_dir):
        shutil.rmtree(base_dir)
        print('  Diretório de dados temporários {} removido com sucesso.'.format(base_dir))

    if os.path.exists(base_dir_filt):
        shutil.rmtree(base_dir_filt)
        print('  Diretório de dados temporários {} removido com
        sucesso.'.format(base_dir_filt))

    if torch.cuda.is_available():
        torch.cuda.empty_cache()
```

A domain far more distant than that of digits

In the transfer learning experiment between handwritten digits (domains A and B), the source and target tasks shared very similar visual statistics: both were grayscale strokes on a uniform background. Here, the distance between domains is considerably greater — YOLO was pretrained on **natural color photographs** from COCO (people, animals, vehicles, everyday objects), and the target task consists of **synthetic geometric shapes, with solid colors and well-defined contours**, lacking texture, lighting, or complex backgrounds.
Even so, transfer learning remains advantageous: the early layers of a detector trained on COCO learn generic filters — edge, corner, and contrast-region detectors — that remain useful for delimiting the contour of a triangle or a star, even though the final visual content is quite distinct. This is why allowing fine-tuning of all layers, combined with consistent preprocessing between training and inference to mitigate salt-and-pepper noise, enables high accuracy rates in detecting objects from the new domain.

Using a Real Dataset

The *pipeline* above was built entirely around the **YOLO annotation format** (class, x_{centro}, y_{centro}, width, height, normalized by the image's width and height) precisely so that it can be reused without modifications if the reader has access to a set of real images annotated in the same way — for example, a set of images of photographed or rendered geometric objects, each with a corresponding `.txt` file in the same format used here. To do so, one would only need to:

1. Organize the real images in `shapes_dataset/images/train` and `shapes_dataset/images/val`, and the corresponding annotation `.txt` files in the `labels/train` and `labels/val` folders (one annotation file per image, same base name, `.txt` extension);
2. Adjust the `data.yaml` file if the number or names of the classes differ;
3. Run the same training, fine-tuning, and visualization cells already presented, without any other code modifications.

This separation between **data generation/organization** and **model training** is, in practice, the reason why standardized annotation formats (such as YOLO's) are so widely adopted: they allow swapping the input dataset — synthetic for real, one domain for another — while keeping all the rest of the transfer learning *pipeline* unchanged.

9.5.3 Object Segmentation

The same transfer learning principle also underpins image segmentation architectures, in which a pretrained feature extractor provides general visual representations, while a specialized head performs dense, pixel-wise classification.

The following sections present **DeepLabV3** as an example of a pretrained segmenter used directly for inference, and then the **U-Net** architecture, trained from scratch and compared with a classic morphological baseline.

9.5.3.1 DeepLabV3: Pre-trained Segmenter

In semantic segmentation, the objective is not limited to locating objects through bounding boxes. The network assigns a class to each pixel of the image, producing a label map with the same resolution as the input. Architectures such as **DeepLabV3**, with a **ResNet-50** backbone, utilize a pre-trained feature extractor and a specialized head to perform this dense classification.

1. **Loading and inference:** The model `deeplabv3_resnet50(weights="DEFAULT")` loads pre-trained weights on the **Pascal VOC** dataset, which defines 21 segmentation classes. The code converts the image into a tensor, adds the batch dimension (`unsqueeze(0)`), and performs inference.

2. **Class map:** The model's output has dimensions $(1, 21, H, W)$, containing a value for each class at every pixel. The operation `.argmax(dim=1)` selects the class with the highest response at each position, generating a two-dimensional label matrix with dimensions (H, W).

3. **Visualization:** The code converts the label map into the format expected by `mm.show()`, which displays the segmentation result in Figure 9.33.

```
# 1. Image loading (returns numpy.ndarray)
url_imagem = "https://raw.githubusercontent.com/pytorch/hub/master/images/dog.jpg"
url_imagem = "http://images.cocodataset.org/val2017/000000039769.jpg"
img = mm.read(url_imagem)

# 2. Loading the pre-trained DeepLabV3 model in evaluation mode
modelo_segmentacao = deeplabv3_resnet50(weights="DEFAULT").eval()

# 3. Running inference without gradient computation
with torch.no_grad():
    tensor_entrada = to_tensor(img).unsqueeze(0)  # Format (1, C, H, W)
    saida = modelo_segmentacao(tensor_entrada)["out"]

    # Selecting the class with the highest probability per pixel (argmax along the channel
    axis)
    mapa_classes = saida.argmax(dim=1).squeeze(0).byte().cpu().numpy()

# 4. Displaying the semantic segmentation map
mm.show(
    [img,mapa_classes],
    title=["Original image","Semantic Segmentation (DeepLabV3)"]
)
```

Figure 9.33

9.5.3.2 Semantic Segmentation with U-Net Architecture

The previous subsection presented a pretrained model (*DeepLabV3*) that directly produces a class label per *pixel*. This section completes the sequence of practical projects—classification and detection—by addressing semantic segmentation implemented and trained from scratch with the **U-Net**, the reference architecture introduced by Ronneberger (2015).

In classification, the final feature map was flattened into a vector, discarding spatial information in favor of a single label per image. The U-Net, in turn, produces an output with the **same spatial resolution as the input**: a two-dimensional map in which each *pixel* receives its own classification. This requirement—preserving high-resolution spatial detail while building semantic context in deep layers—motivates the encoder-decoder architecture with shortcut connections (*skip connections*).

The scenario used simulates the segmentation of nodules in synthetic medical exams: grayscale images contain a circular region ("nodule") overlaid on a background, both contaminated by Gaussian noise and with very similar intensity means—an intentional low-contrast challenge, ideal for demonstrating the benefit of spatial learning compared to pointwise thresholding.

Block 1: Synthetic Nodule Set Generator and Initial Display

The following generator produces (image, mask) pairs: the image contains a circular region of intensity slightly higher than that of the background, both affected by the same standard deviation of Gaussian noise. The binary mask precisely delimits the nodule region and serves as the reference ground truth.

1. **Nodule Construction:** The function `gera_imagem_com_nodulo` overlays the nodule onto the background on a 64×64 matrix and applies Gaussian noise.

2. **Visualization with `mm.show`:** Figure 9.34 illustrates the first two samples and their respective masks.

```
TAM_IMG = 64


def gera_imagem_com_nodulo(
    tam=TAM_IMG,
    raio_min=7,
    raio_max=15,
    media_fundo=95,
    media_nodulo=118,
    sigma_ruido=26,
    rng=None,
):
    rng = rng or np.random.default_rng()
    fundo = rng.normal(media_fundo, sigma_ruido, (tam, tam))
    nodulo = rng.normal(media_nodulo, sigma_ruido, (tam, tam))
    mascara = np.zeros((tam, tam), dtype=np.uint8)

    raio = int(rng.integers(raio_min, raio_max))
    cx = int(rng.integers(raio + 4, tam - raio - 4))
    cy = int(rng.integers(raio + 4, tam - raio - 4))

    cv2.circle(mascara, (cx, cy), raio, 255, -1)
    imagem = np.clip(np.where(mascara > 0, nodulo, fundo), 0, 255).astype(
        np.uint8
    )
    return imagem, mascara


# Generation of training and validation sets
rng_dados = np.random.default_rng(42)
N_TREINO, N_VAL = 160, 40

imgs_treino, masks_treino = zip(
    *[gera_imagem_com_nodulo(rng=rng_dados) for _ in range(N_TREINO)]
)
imgs_val, masks_val = zip(
    *[gera_imagem_com_nodulo(rng=rng_dados) for _ in range(N_VAL)]
)

print(
    f"Training set: {N_TREINO} images | Validation set: {N_VAL} images\n"
)

# Display of initial samples
mm.show(
    [imgs_treino[0], masks_treino[0], imgs_treino[1], masks_treino[1]],
    titles=["Image 1", "Mask 1", "Image 2", "Mask 2"],
    cols=4,
    figsize=(11, 3),
)
```

```
Training set: 160 images | Validation set: 40 images
```

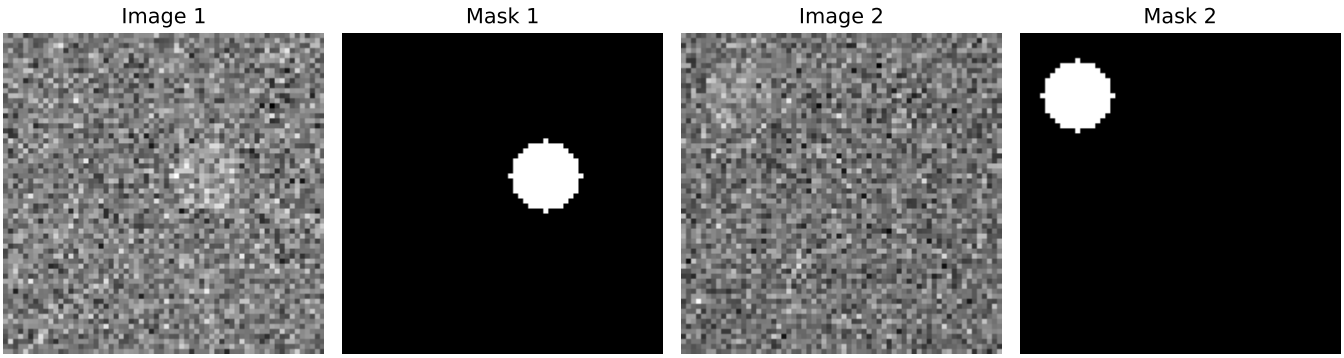


Figure 9.34: Samples from the synthetic nodule dataset: grayscale image under noise and its respective reference binary mask shown.

Block 2: Classical Baseline (Filtering, Otsu, and Morphology)

Before employing the U-Net, the performance of a classical morphological *pipeline* built with the `morph` library is evaluated: a **Gaussian smoother** (`mm.blur`), an **Otsu thresholding** (`mm.threshold`), and a **morphological opening** (`mm.open`) to eliminate isolated noise.

1. **Intersection over Union (IoU) Metric:** The function `iou_mascaras` computes the degree of *pixel-by-pixel* overlap between the prediction and the ground truth mask.
2. **Execution and Comparison:** Figure 9.35 displays the result of classical segmentation on a test image, demonstrating the limitations of the global threshold under low contrast.

```
def iou_mascaras(predita, referencia):
    p, r = predita > 0, referencia > 0
    intersecao = np.logical_and(p, r).sum()
    uniao = np.logical_or(p, r).sum()
    return intersecao / uniao if uniao else 1.0


def segmenta_classico(imagem, elemento_estrutural):
    suavizada = mm.blur(imagem, 7)
    binaria = mm.threshold(suavizada)
    return mm.open(binaria, elemento_estrutural)


elemento_estrutural = mm.sedisk(5)
ious_classico = [
    iou_mascaras(segmenta_classico(img, elemento_estrutural), mask)
    for img, mask in zip(imgs_val, masks_val)
]
iou_classico_medio = float(np.mean(ious_classico))
print(
    f"Average IoU (classical baseline) on validation: {iou_classico_medio:.4f}\n"
)

predicao_classica_exemplo = segmenta_classico(imgs_val[0], elemento_estrutural)

mm.show(
    [imgs_val[0], masks_val[0], predicao_classica_exemplo],
    titles=["Image", "Reference Mask", "Classical Prediction"],
    cols=3,
    figsize=(9, 3.2),
)
```

```
Average IoU (classical baseline) on validation: 0.7516
```

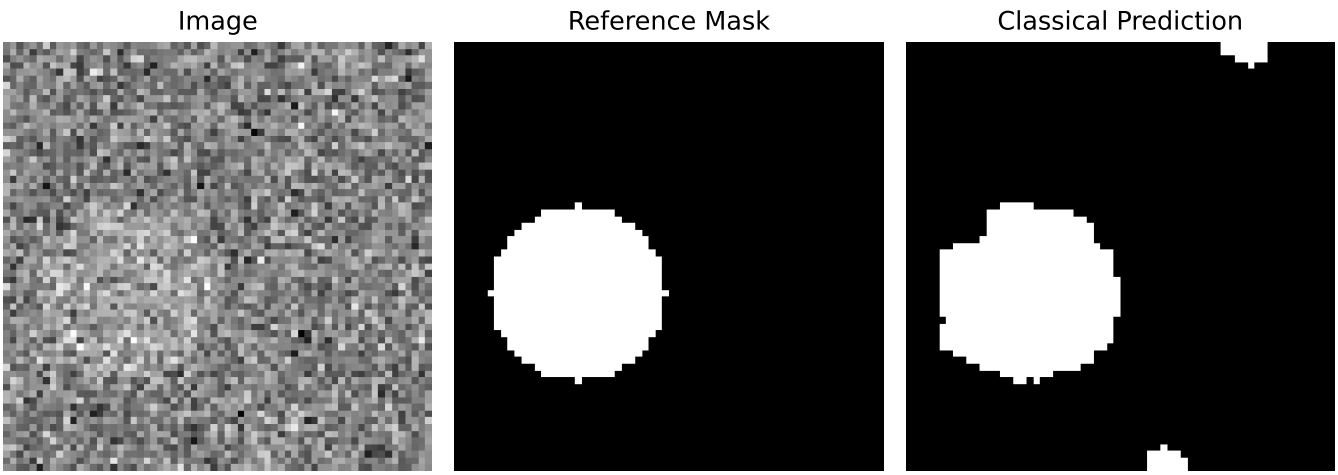


Figure 9.35: Classical segmentation baseline: smoothing, Otsu thresholding, and morphological opening displayed.

Block 3: Construction of the U-Net Architecture and Loss Functions

The U-Net is a "U"-shaped architecture (hence the name), specifically designed for image segmentation. It consists of two pathways that work together:

- ↓ **Encoder:** descends through the image, reducing spatial resolution at each step while extracting increasingly abstract features (edges → textures → shapes → context).
- **Decoder:** ascends back, reconstructing the original resolution through transposed convolutions (*upsampling*), until it generates a mask of the same size as the input image.

The element that makes U-Net special is the **skip connections**: they carry feature maps from the encoder directly to the corresponding stage of the decoder, at the same resolution. This prevents fine details—such as contours and edges—from being lost during spatial compression.

General architecture flow:

```
Input


Encoder (Conv → Conv → Pool) × 3

      skip connections reinject high-resolution maps

Bottleneck


Decoder (Upsample → Concat → Conv → Conv) × 3


Output 1×1 (logits)
```

Main components:

1. **Base Convolutional Block (`BlocoConv`)** The fundamental unit repeated throughout the network. It applies two 3×3 convolutions in sequence, each followed by ReLU activation, with padding that preserves the spatial dimensions of the input. This block appears in both the encoder and the decoder.

2. **Transposed Convolution (`nn.ConvTranspose2d`)** This is the operation responsible for *upsampling* in the decoder: instead of reducing spatial resolution (as `MaxPool2d` does in the encoder), it increases it, learning the necessary weights to "undo" the compression and gradually recover the original image size.

3. **Combined Loss (BCE + Dice)** The `perda_segmentacao` function sums two complementary metrics:

   - **Binary Cross-Entropy (BCE):** evaluates pixel-by-pixel accuracy.
   - **Dice Coefficient:** evaluates the overall overlap between the predicted and ground-truth masks.

   Together, they balance local precision with fidelity of the segmented shape as a whole.

```
class BlocoConv(nn.Module):

    def __init__(self, canais_entrada, canais_saida):
        super().__init__()
        self.rede = nn.Sequential(
            nn.Conv2d(canais_entrada, canais_saida, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(canais_saida, canais_saida, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
        return self.rede(x)


class UNetCompacta(nn.Module):

    def __init__(self, canais_entrada=1, base=8):
        super().__init__()
        self.enc1 = BlocoConv(canais_entrada, base)
        self.enc2 = BlocoConv(base, base * 2)
        self.enc3 = BlocoConv(base * 2, base * 4)
        self.pool = nn.MaxPool2d(2)

        self.fundo = BlocoConv(base * 4, base * 8)

        self.up3 = nn.ConvTranspose2d(
            base * 8, base * 4, kernel_size=2, stride=2
        )
        self.dec3 = BlocoConv(base * 8, base * 4)
        self.up2 = nn.ConvTranspose2d(
            base * 4, base * 2, kernel_size=2, stride=2
        )
        self.dec2 = BlocoConv(base * 4, base * 2)
        self.up1 = nn.ConvTranspose2d(
            base * 2, base, kernel_size=2, stride=2
        )
        self.dec1 = BlocoConv(base * 2, base)

        self.saida = nn.Conv2d(base, 1, kernel_size=1)

    def forward(self, x):
        e1 = self.enc1(x)
        e2 = self.enc2(self.pool(e1))
        e3 = self.enc3(self.pool(e2))
        f = self.fundo(self.pool(e3))

        d3 = self.dec3(torch.cat([self.up3(f), e3], dim=1))
        d2 = self.dec2(torch.cat([self.up2(d3), e2], dim=1))
        d1 = self.dec1(torch.cat([self.up1(d2), e1], dim=1))
        return self.saida(d1)


def para_tensores(imagens, mascaras):
```

```
    X = (
        torch.tensor(np.stack(imagens), dtype=torch.float32).unsqueeze(1)
        / 255.0
    )
    Y = (
        torch.tensor(np.stack(mascaras), dtype=torch.float32).unsqueeze(1)
        / 255.0
    )
    return X, Y


def perda_dice(logits, alvo, eps=1e-6):
    probs = torch.sigmoid(logits)
    intersecao = (probs * alvo).sum(dim=(1, 2, 3))
    uniao = probs.sum(dim=(1, 2, 3)) + alvo.sum(dim=(1, 2, 3))
    dice = (2 * intersecao + eps) / (uniao + eps)
    return 1 - dice.mean()


def perda_segmentacao(logits, alvo):
    return nn.functional.binary_cross_entropy_with_logits(
        logits, alvo
    ) + perda_dice(logits, alvo)
```

Block 4: U-Net Training and Performance Evaluation

Training is run for 35 epochs using the *Adam* optimizer. At each epoch, the Training Loss and the mean IoU index on the validation set are monitored.

1. **Training Loop:** Parameters are updated with *mini-batches* of 16 samples.
2. **Graphical Evolution:** Figure 9.36 displays the graph with the progress of loss and IoU.

```
modelo_unet = UNetCompacta()
print(
    f"Trainable U-Net parameters: {sum(p.numel() for p in modelo_unet.parameters())}"
)

X_treino_unet, Y_treino_unet = para_tensores(imgs_treino, masks_treino)
X_val_unet, Y_val_unet = para_tensores(imgs_val, masks_val)

otimizador_unet = optim.Adam(modelo_unet.parameters(), lr=1e-3)
n = X_treino_unet.size(0)
tam_lote = 16
epocas_unet = 35
historico_perda_unet, historico_iou_unet = [], []

for epoca in range(epocas_unet):
    if epoca % 5 == 0:  # Prints every 5 epochs
        print(epoca + 1, "/", epocas_unet)
    modelo_unet.train()
    perm = torch.randperm(n)
    perda_epoca = 0.0

    for i in range(0, n, tam_lote):
        idx = perm[i : i + tam_lote]
        otimizador_unet.zero_grad()
        logits = modelo_unet(X_treino_unet[idx])
        perda = perda_segmentacao(logits, Y_treino_unet[idx])
        perda.backward()
        otimizador_unet.step()
```

```
        perda_epoca += perda.item() * len(idx)

    modelo_unet.eval()
    with torch.no_grad():
        predicao_val = (torch.sigmoid(modelo_unet(X_val_unet)) > 0.5).float()
        intersecao = (predicao_val * Y_val_unet).sum(dim=(1, 2, 3))
        uniao = ((predicao_val + Y_val_unet) > 0).float().sum(dim=(1, 2, 3))
        iou_epoca = (intersecao / uniao.clamp(min=1e-6)).mean().item()

    historico_perda_unet.append(perda_epoca / n)
    historico_iou_unet.append(iou_epoca)

iou_unet_final = historico_iou_unet[-1]
print(f"Final average IoU of U-Net on validation: {iou_unet_final:.4f}")

# Training evolution chart
r = mm.showTrainCurves(historico_perda_unet, historico_iou_unet,
        titulo="Treinamento da U-Net - Segmentação de Nódulos Sintéticos",
        subtitulo="Perda no conjunto de treino e IoU médio no conjunto de validação \
            ao longo de 35 épocas")
```

```
Trainable U-Net parameters: 120681
1 / 35
6 / 35
11 / 35
16 / 35
21 / 35
26 / 35
31 / 35
Final average IoU of U-Net on validation: 0.8876
```

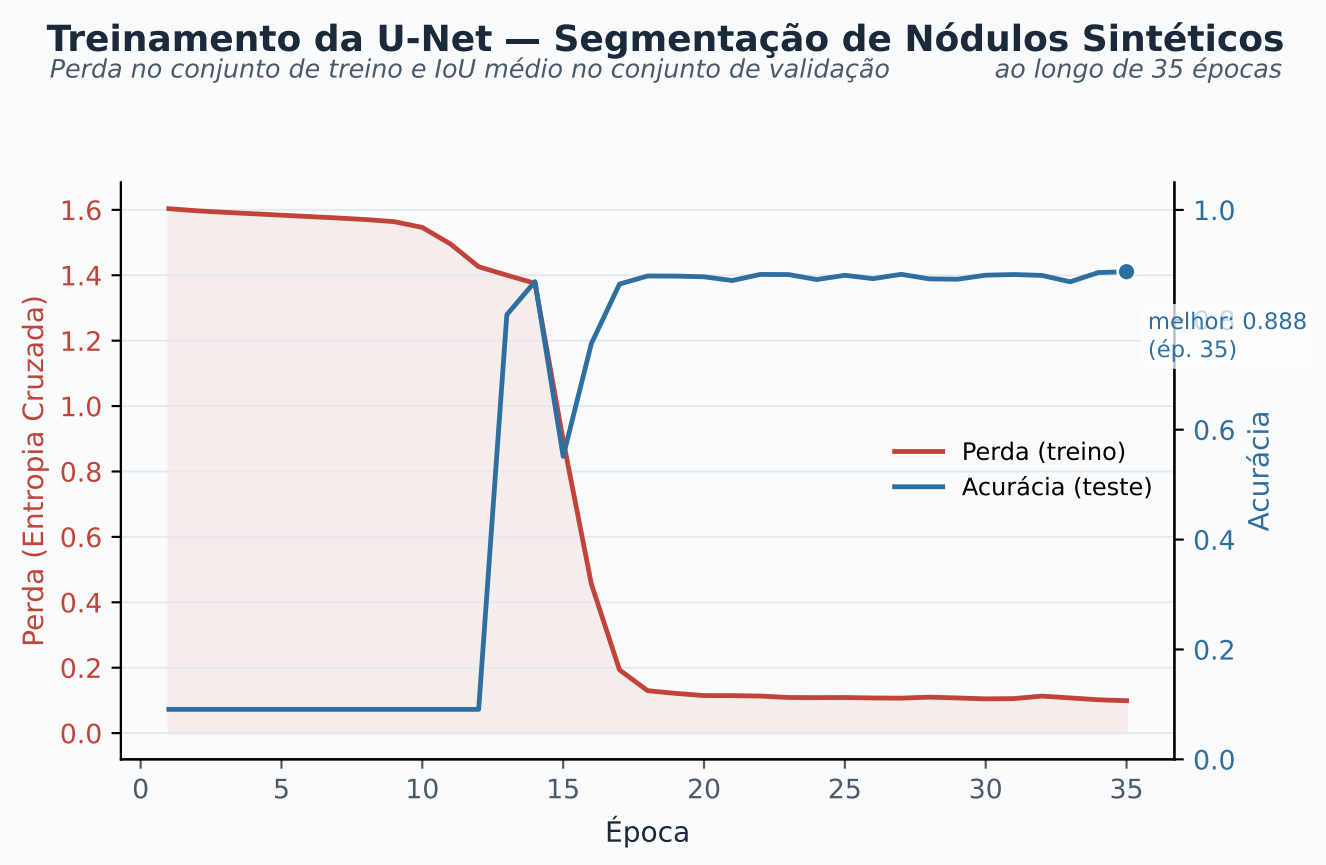


Figure 9.36: Evolution of compact U-Net training: loss reduction and increase in the average IoU index on the validation set over 35 epochs.

Block 5: Quantitative and Qualitative Comparison (Classic vs. U-Net)

The comparison between the classic approach and the U-Net highlights the superiority of representation learning in low-contrast scenarios.

1. **Metrics Panel:** The bar chart in Figure 9.37 contrasts the mean IoU of both methods on validation.
2. **Qualitative Visualization:** The visual comparison across three samples demonstrates how skip connections recover the nodule contour even under pronounced noise.

```
# 1. Comparison
print(f"Mean IoU (classical Smoothing + Otsu): {iou_classico_medio:.4f}")
print(f"Mean IoU (U-Net): {iou_unet_final:.4f}")

# 2. Qualitative side-by-side display in 3 samples via mm.show
imgs_comparativas = []
titulos_comparativos = []

for i in range(3):
    with torch.no_grad():
        pred_unet = (
            (torch.sigmoid(modelo_unet(X_val_unet[i : i + 1])) > 0.5)
            .float()
            .squeeze()
            .numpy()
            * 255
        )

    pred_classico = segmenta_classico(imgs_val[i], elemento_estrutural)
```

```
    imgs_comparativas.extend(
        [imgs_val[i], masks_val[i], pred_classico, pred_unet.astype(np.uint8)]
    )

    t_prefix = f"Amostra {i+1}"
    titulos_comparativos.extend(
        [
            f"{t_prefix}: Imagem",
            f"{t_prefix}: Referência",
            f"{t_prefix}: Clássico",
            f"{t_prefix}: U-Net",
        ]
    )

mm.show(
    imgs_comparativas,
    titles=titulos_comparativos,
    cols=4,
    figsize=(11, 7.5),
)
```

```
Mean IoU (classical Smoothing + Otsu): 0.7516
Mean IoU (U-Net): 0.8876
```

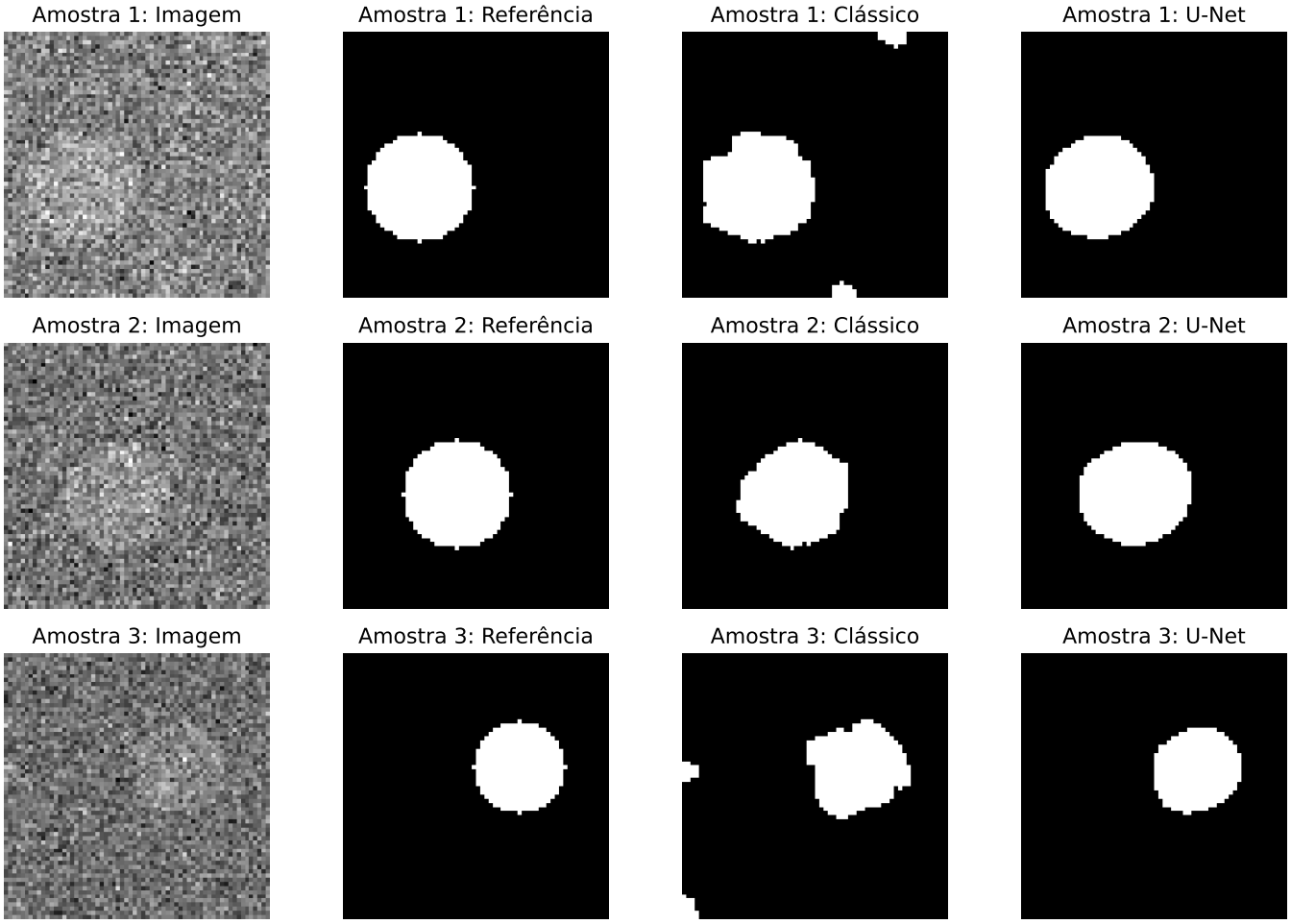


Figure 9.37: Quantitative comparison (mean IoU) and qualitative comparison between the classical approach and the U-Net trained on three validation samples.

Why Does U-Net Outperform Fixed Thresholding?

Otsu thresholding applies a global cutoff value to local intensity. When the mean difference between the nodule and the background is small relative to Gaussian noise, this rule makes systematic errors at the edges.
U-Net overcomes this limitation by combining the broad semantic context extracted by the encoder with the fine spatial details preserved by the skip connections. This enables the identification of the nodule's presence and the precise delineation of its contours, even under intense noise.

9.5.4 Data Engineering for CV (*Roboflow*)

A U-Net can be trained from images annotated on data engineering platforms for Computer Vision (CV), such as *Roboflow*. These platforms allow you to organize datasets, perform annotations, apply preprocessing steps and *data augmentation*, train models, and export data in various formats. In this section, however, the example revisits **geometric object detection**, using datasets already presented in this book.

Dependency on Internet connection and API key

The cells in this section require an Internet connection and a free *Roboflow* API key (`app.roboflow.com`).
In the project's *Workspace*, access **§ → Roboflow API** and copy the **Private API Key**.
Create the file `chave_roboflow.txt` in this folder, containing **only the key**, without quotation marks.
A template for this file is available in `chave_roboflow.txt.exemplo`.
Add `chave_roboflow.txt` to the `.gitignore` file, as it contains an access credential that should not be versioned or shared.

9.5.4.1 Object Detection using *Roboflow*

Roboflow allows inference on local images, enabling the assessment of the hosted model's performance in identifying objects of interest.

In addition to inference, the *dataset* can be exported in the `png-mask-semantic` format, where each image is accompanied by a semantic segmentation mask. In this mask, each pixel represents the class to which the corresponding object belongs. The image-mask pairs are used as training data for `UNetCompacta`.

9.5.4.2 Connection, *download*, and verification of the *dataset*

The code establishes a connection to the `mctest` *Workspace* and the `geometric-test00` project, version 6, and performs the *download* of the *dataset* to `dados/datasetRoboFlow`. It then verifies the *shape* of the images in each *split*. The verification function is reused later in the section.

```
from roboflow import Roboflow
from pathlib import Path
from PIL import Image
from collections import Counter

def contar_shapes(raiz, splits=("train", "valid", "test")):
    """Counts the shape (height, width, channels) of images per split."""
    for split in splits:
        pasta = raiz / split / "images"
        if not pasta.exists():
            print(f"{split}: folder not found")
            continue

        shapes = Counter()
        for arquivo in pasta.iterdir():
            if arquivo.is_file():
```

```
                with Image.open(arquivo) as img:
                    shapes[(img.height, img.width, len(img.getbands()))] += 1

        txt = ", ".join(f"{s}: {n}" for s, n in shapes.items())
        print(f"{split}: {txt}")


chave = Path("chave_roboflow.txt")

if not chave.exists():
    print("Roboflow key not found: chave_roboflow.txt")
else:
    with open(chave) as f:
        api_key = f.read().strip()

    # Project:
    # https://app.roboflow.com/mctest/geometric-test00/models
    # geometric-test00/6

    rf = Roboflow(api_key=api_key)
    projeto = rf.workspace("mctest").project("geometric-test00")
    versao = projeto.version(6)

    print(
        f"ID: {versao.version} | Name: {versao.name} | "
        f"Images: {versao.images}"
    )

    raiz = Path("dados/datasetRoboFlow")
    versao.download("yolov8", location=str(raiz))

    print("Dataset:", raiz)
    contar_shapes(raiz)
```

9.5.4.3 Downloading the dataset reproducibly (alternative)

As an alternative to the dataset obtained via *Roboflow*, one may use a dataset available in a GitHub repository, also organized into the same *splits* and containing the same object classes. The data sets, however, are not identical: the GitHub images have a resolution of `608×608` pixels, while the images exported by *Roboflow* have `640×640` pixels.

The code below performs the *download* of the GitHub dataset, if it is not already available locally, and reuses `contar_shapes` to verify the dimensions of the images in each *split*.

```
import os

if not os.path.exists("dados/dataset"):
    cmd = (
        "git clone --no-checkout --depth 1 --filter=blob:none "
        "https://github.com/fzampirolli/pdi-vc.git tmp_repo && "
        "cd tmp_repo && git sparse-checkout set all/cap09/dados/dataset "
        "&& git checkout && cd .. && mkdir -p dados && "
        "cp -r tmp_repo/all/cap09/dados/dataset dados/dataset && "
        "rm -rf tmp_repo"
    )
    !{cmd}

if not chave.exists():
    print("Chave do Roboflow não encontrada: chave_roboflow.txt")
else:
```

```
versao_recente = projeto.versions()[-1]
print(f"Versão mais recente: {versao_recente.version.split('/')[-1]}")

contar_shapes(Path("dados/dataset"))
```

9.5.4.4 Comparing one image from each *dataset*

The two *datasets* have images with different resolutions: `608×608` on GitHub and `640×640` on *Roboflow*. For a direct visual comparison, the images are resized to the same dimensions before being displayed side by side (Figure 9.38).

```
import cv2

if not chave.exists():
    print("Roboflow key not found: key_roboflow.txt")
else:
    caminho1 = next((raiz / "train/images").iterdir())
    caminho2 = next((Path("dados/dataset") / "train/images").iterdir())

    img1 = mm.read(str(caminho1))
    img2 = mm.read(str(caminho2))

    # Resizes both to the same size (the smaller of the two)
    largura = min(img1.shape[1], img2.shape[1])
    altura = min(img1.shape[0], img2.shape[0])
    img1_r = cv2.resize(img1, (largura, altura))
    img2_r = cv2.resize(img2, (largura, altura))

    mm.show(
        [img1_r, img2_r],
        title=[f"Roboflow {img1.shape}", f"GitHub {img2.shape}"],
    )
```

Figure 9.38

9.5.4.5 Inference with the trained model

The model trained in version 6 is used to perform inference on a local test image. The prediction considers the confidence and overlap thresholds employed by non-maximum suppression (NMS), and the result is presented in the annotated image of Figure 9.39.

9.5.4.6 Evaluating predictions with IoU and class

Roboflow returns each box with the center coordinates (`x`, `y`) in *pixels* and the predicted class, while local labels (`dados/dataset/test/labels/00001.txt`) follow the YOLO format, with center and dimensions normalized to the range `[0, 1]`. Before comparison using `mm.IoU`, the boxes must be converted to the same format, with top-left corner coordinates and dimensions expressed in *pixels*.

A prediction is only considered correct when the predicted class matches the class of the ground-truth box and its *Intersection over Union* (IoU) is greater than or equal to the defined threshold, adopting, in this example, `0.5` (50%) as the default value.

> **!** The *Roboflow* `class_id` does not correspond to the index of the local labels
>
> During export, *Roboflow* reorders the classes in **alphabetical order** in the `data.yaml`, regardless of the order used in the original project, preserved in the GitHub `data.yaml`. Thus, `class_id = 0` corresponds to `Circulo` in the API response, while the same index corresponds to `Triangulo` in the

```python
# version.model is deprecated; use version.models()

if not chave.exists():
    print("Roboflow key not found: chave_roboflow.txt")
else:
    modelo = versao.models()[0]

    img_path = "dados/dataset/test/images/00001.jpg"
    pred = modelo.predict(img_path, confidence=40, overlap=30)
    resp = pred.json()  # includes the detected boxes in resp["predictions"]

    altura, largura, _ = mm.read(img_path).shape
    print("Test image dimensions:", (altura, largura))

    pred.save("resultado.jpg")  # annotated image

    mm.show(
        mm.read("resultado.jpg"),
        title="Result of inference with the Roboflow model",
        figsize=(6, 6)
    )
```

Figure 9.39

local labels.
Therefore, the comparison must be made by **class name** (`p["class"]`), subsequently converting it to the corresponding index in the local list. The `class_ids` should not be compared directly.

```python
# resp, width and height were defined in the previous cell

# Order of classes used in the local labels (data/dataset/*/labels/*.txt)
CLASSES_LOCAIS = ["Triangulo", "Quadrado", "Pentagono", "Hexagono",
                  "Heptagono", "Circulo", "Elipse"]

def predicao_correta(pred: tuple, real: tuple, limiar: float = 0.5) -> bool:
    """True if same class and mm.IoU(pred_box, real_box) >= threshold."""
    classe_pred, caixa_pred = pred
    classe_real, caixa_real = real
    return classe_pred == classe_real and mm.IoU(caixa_pred, caixa_real) >= limiar

def caixa_roboflow(p: dict) -> tuple:
    """Converts Roboflow prediction to (class, (x, y, w, h))."""
    caixa = (p["x"] - p["width"] / 2, p["y"] - p["height"] / 2,
             p["width"], p["height"])
    # uses the class name (not the class_id!) to match the local order
    classe = CLASSES_LOCAIS.index(p["class"])
    return (classe, caixa)

def carrega_labels_yolo(caminho_txt: str, largura: int, altura: int) -> list:
    """Reads YOLO labels (normalized) and converts to (class, (x, y, w, h))."""
    caixas = []
    with open(caminho_txt) as f:
        for linha in f:
            classe, xc, yc, w, h = map(float, linha.split())
            w_px, h_px = w * largura, h * altura
            x_px = xc * largura - w_px / 2
            y_px = yc * altura - h_px / 2
```

```
            caixas.append((int(classe), (x_px, y_px, w_px, h_px)))
    return caixas

if not chave.exists():
    print("Roboflow key not found: roboflow_key.txt")
else:
    caixas_pred = [caixa_roboflow(p) for p in resp["predictions"]]
    caixas_real = carrega_labels_yolo(
        "dados/dataset/test/labels/00001.txt", largura, altura
    )

    acertos = sum(
        any(predicao_correta(cp, cr, limiar=0.5) for cr in caixas_real)
        for cp in caixas_pred
    )
    print(f"{hits}/{len(pred_boxes)} correct predictions (class + IoU >= 50%)")
```

To visualize the result, each prediction is drawn over the image: green when it is a hit (class + IoU threshold) and red when it is a miss, as shown in Figure 9.40.

```
import cv2

VERDE, VERMELHO = (0, 255, 0), (255, 0, 0)

if not chave.exists():
    print("Roboflow key not found: chave_roboflow.txt")
else:
    img_acertos = mm.read(img_path).copy()

    if not chave.exists():
        print("Roboflow key not found: chave_roboflow.txt")
    else:
        for cp in caixas_pred:
            classe_pred, (x, y, w, h) = cp
            correta = any(predicao_correta(cp, cr, limiar=0.5) for cr in caixas_real)
            cor = VERDE if correta else VERMELHO

            p1, p2 = (int(x), int(y)), (int(x + w), int(y + h))
            cv2.rectangle(img_acertos, p1, p2, cor, 2)
            cv2.putText(img_acertos, str(classe_pred), (p1[0], p1[1] - 5),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5, cor, 2)

    mm.show(
        img_acertos,
        title=f"{acertos}/{len(caixas_pred)} correct predictions "
            f"(class + IoU >= 50%)",
        figsize=(6, 6)
    )
```

Figure 9.40

9.5.5 Geometric and Integrated Applications

The chapter concludes by integrating two pillars of CV: **projective geometry** (studied in Chapters 6 and 8) and **deep learning**. The combination of these approaches supports practical real-world applications, as illustrated below.

9.5.5.1 Augmented Reality with Markers and Homography

Imagine a camera pointed at a table where someone has placed a small ArUco marker. Depending on the camera's angle, that marker appears **rotated, tilted, in perspective** — never perfectly square. It is precisely this distortion that homography knows how to "read" and undo (or, in our case, replicate for a new image).

The complete workflow of a marker-based AR application follows three steps:

1. **Setup:** the marker is inserted into a real scene, undergoing a perspective transformation (simulating the camera's angle).
2. **Detection:** the algorithm locates the marker in the scene and recovers the exact coordinates of its 4 corners using `cv2.aruco.ArucoDetector`.
3. **Replacement:** with the homography between the "ideal" marker and the "detected" marker, a **new virtual image** is projected exactly over the marker's area — as if it had transformed into a window for other content, as shown in Figure 9.41.

> **Note**
>
> **Important technical detail:** ArUco requires a white margin around the pattern (the "quiet zone") for the detector to distinguish the marker from the background. Therefore, the code below adds a border with `cv2.copyMakeBorder` and uses `cv2.INTER_NEAREST` interpolation when warping the image, preventing rotation from blurring the black and white squares and hindering detection.

```
# 1. Generation of the synthetic ArUco marker, already with white margin (quiet zone)
# → this margin is essential for the detector to be able to "see" the marker
dic = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_4X4_50)
aruco_bruto = cv2.aruco.generateImageMarker(dic, 7, 200)
aruco_gray = cv2.copyMakeBorder(
    aruco_bruto, 40, 40, 40, 40, cv2.BORDER_CONSTANT, value=255
)  # 200 -> 280px, with 40px of white border on each side
aruco = cv2.cvtColor(aruco_gray, cv2.COLOR_GRAY2BGR)
lado = aruco.shape[0]  # 280

# 2. Real background scene, using an example image from skimage.data
fundo = cv2.cvtColor(skdata.coffee(), cv2.COLOR_RGB2BGR)
cena = cv2.resize(fundo, (640, 480))

# Corners of the "front-facing" marker (src) and its rotated/tilted position in the scene
(dst)
src = np.float32([[0, 0], [lado, 0], [lado, lado], [0, lado]])
dst = np.float32([[190, 160], [420, 70], [470, 330], [150, 370]])  # rotation + perspective

# 3. Projects the marker (with sharp edges) onto the real scene
H_cena, _ = cv2.findHomography(src, dst)
mask_cena = cv2.warpPerspective(
    np.full((lado, lado), 255, np.uint8), H_cena, (640, 480),
    flags=cv2.INTER_NEAREST,
)
cena[mask_cena > 0] = cv2.warpPerspective(
    aruco, H_cena, (640, 480), flags=cv2.INTER_NEAREST
)[mask_cena > 0]

# 4. Detection of the marker within the scene (as a camera would)
det = cv2.aruco.ArucoDetector(dic, cv2.aruco.DetectorParameters())
corners, ids, _ = det.detectMarkers(cena)

assert ids is not None and len(corners) > 0, \
    "Marcador não detectado - confira iluminação/contraste da cena."
```

```
# 5. Virtual image (another image from skimage.data) that will "replace" the marker
# Note: we use the INNER square of the marker (without the margin) as the projection area,
# so the homography of the virtual image uses the original 'src' (200x200), not the 'side'
with border
src_interno = np.float32([[0, 0], [200, 0], [200, 200], [0, 200]])
virtual = cv2.resize(
    cv2.cvtColor(skdata.camera(), cv2.COLOR_RGB2BGR), (200, 200)
)

# The detected corners correspond to the marker WITH the margin (280x280),
# so we recalculate H_ra using 'src' with margin, to keep the projection consistent
H_ra, _ = cv2.findHomography(src, corners[0][0])

# The detected homography is applied to the virtual image (resized to 'side')
virtual_grande = cv2.resize(virtual, (lado, lado))
ra = cena.copy()
mask_ra = cv2.warpPerspective(
    np.full((lado, lado), 255, np.uint8), H_ra, (640, 480), flags=cv2.INTER_NEAREST
)
ra[mask_ra > 0] = cv2.warpPerspective(
    virtual_grande, H_ra, (640, 480), flags=cv2.INTER_NEAREST
)[mask_ra > 0]

# Display: scene with marker vs. scene with Augmented Reality applied
mm.show(
    [cv2.cvtColor(cena, cv2.COLOR_BGR2RGB), cv2.cvtColor(ra, cv2.COLOR_BGR2RGB)],
    titles=["Marker in the Real Scene (rotated)", "Virtual Overlay via Homography"],
    cols=2,
    figsize=(9, 4),
)
```

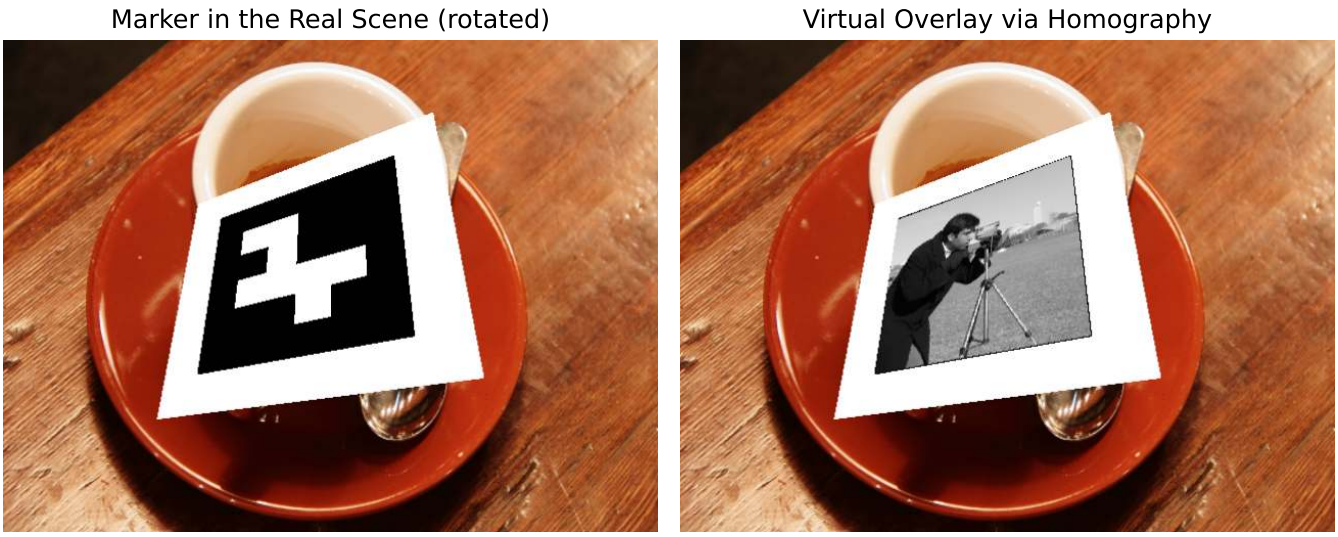


Figure 9.41: Augmented reality based on ArUco marker: marker inserted in real scene and rotated, detected and replaced by virtual image via homography.

Pipeline summary:

Step	What it does	Key function
1. Generation	ArUco marker with white border	`generateImageMarker` + `copyMakeBorder`

Step	What it does	Key function
2. Placement	Inserts distorted marker into the scene	`findHomography` + `warpPerspective`
3. Detection	Locates marker and returns corners	`ArucoDetector.detectMarkers`
4. Replacement	Projects virtual image onto the marker	`findHomography` + `warpPerspective`

💡 **Lesson:** The detector "finding nothing" is common in CV. Always ask: **"did I provide enough contrast and space?"** — this applies to ArUco, QR codes, and facial recognition.

9.5.5.2 Photogrammetry and Scale Reference

Photogrammetry enables the estimation of physical dimensions of objects from digital images. To achieve this, a reference object with known dimensions is placed in the same scene as the object of interest. This procedure establishes a relationship between distances measured in *pixels* and their corresponding dimensions in the real world.

Consider, for example, a credit card, whose dimensions follow the international standard ISO/IEC 7810. Since its width is exactly **8.56 cm**, it suffices to determine how many *pixels* this width occupies in the image to calculate the conversion factor between *pixels* and centimeters. If the card corresponds to 140 *pixels*, then each *pixel* will represent approximately **0.061 cm**. This same scale factor can be applied to estimate the dimensions of any other object located on the same plane of the scene, as illustrated in Figure 9.42.

The procedure can be divided into two main steps:

1. **Segmentation and bounding box:** locate, in the image, both the reference object and the object of interest, using techniques such as color segmentation, thresholding, contour detection, or object detection methods.
2. **Conversion to physical dimensions:** calculate the ratio cm/pixel from the known width of the reference object and use it to convert the measurements of the object of interest from *pixels* to centimeters.

> ℹ Note
>
> **Condition for reliable measurements**
> The conversion between *pixels* and centimeters assumes that the reference object and the object of interest are approximately on the same plane and at the same distance from the camera. Under these conditions, the scale remains practically constant throughout the image. Differences in depth, camera tilt, or lens distortions can introduce errors in the estimated measurements.

```
COR_REFERENCIA = (200, 200, 200)  # Reference card (gray)
COR_OBJETO = (60, 60, 220)  # Target object (red)

# Synthetic scene drawing
cena_medicao = np.full((300, 500, 3), 255, dtype=np.uint8)
cv2.rectangle(
    cena_medicao, (30, 200), (30 + 140, 200 + 88), COR_REFERENCIA, -1
)
cv2.rectangle(cena_medicao, (250, 100), (250 + 220, 100 + 150), COR_OBJETO, -1)


def caixa_delimitadora_por_cor(imagem_bgr, cor_bgr, tolerancia=40):
    diferenca = np.abs(imagem_bgr.astype(int) - np.array(cor_bgr)).sum(axis=2)
    mascara = (diferenca < tolerancia).astype(np.uint8) * 255
    contornos, _ = cv2.findContours(
```

```
        mascara, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
    )
    maior_contorno = max(contornos, key=cv2.contourArea)
    return cv2.boundingRect(maior_contorno)


x_ref, y_ref, w_ref_px, h_ref_px = caixa_delimitadora_por_cor(
    cena_medicao, COR_REFERENCIA
)
x_obj, y_obj, w_obj_px, h_obj_px = caixa_delimitadora_por_cor(
    cena_medicao, COR_OBJETO
)

# Physical scale calculation
LARGURA_REFERENCIA_CM = 8.56
razao_cm_por_px = LARGURA_REFERENCIA_CM / w_ref_px
largura_obj_cm = w_obj_px * razao_cm_por_px
altura_obj_cm = h_obj_px * razao_cm_por_px

# Drawing boxes and estimated measurements
resultado = cena_medicao.copy()
cv2.rectangle(
    resultado, (x_ref, y_ref), (x_ref + w_ref_px, y_ref + h_ref_px), (0, 180, 0), 2
)
cv2.rectangle(
    resultado, (x_obj, y_obj), (x_obj + w_obj_px, y_obj + h_obj_px), (0, 180, 0), 2
)

cv2.putText(
    resultado,
    f"{LARGURA_REFERENCIA_CM:.2f} cm",
    (x_ref, y_ref - 8),
    cv2.FONT_HERSHEY_SIMPLEX,
    0.55,
    (0, 120, 0),
    2,
)

cv2.putText(
    resultado,
    f"{largura_obj_cm:.1f} x {altura_obj_cm:.1f} cm",
    (x_obj, y_obj - 8),
    cv2.FONT_HERSHEY_SIMPLEX,
    0.6,
    (0, 120, 0),
    2,
)

mm.show(
    [
        cv2.cvtColor(cena_medicao, cv2.COLOR_BGR2RGB),
        cv2.cvtColor(resultado, cv2.COLOR_BGR2RGB),
    ],
    titles=[
        "Original Scene",
        "Scale Reference Measurement (Centimeters)",
    ],
    cols=2,
    figsize=(9, 4),
)
```

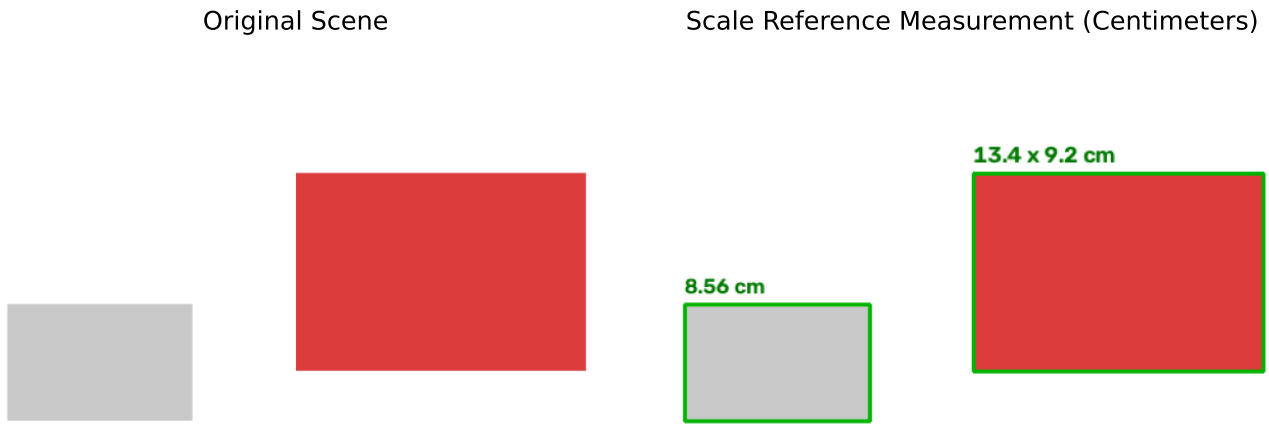


Figure 9.42: Measurement of real physical dimensions using a reference card with a known scale (8.56 cm) rendered. The gray rectangle represents the reference card and the blue rectangle represents the target object. The estimated dimensions of the object are displayed in centimeters.

Pipeline summary:

Step	What it does	Key function
1. Synthetic scene	Draws the reference card and target object with distinct colors	`cv2.rectangle`
2. Segmentation	Isolates each object by color and extracts its contour	`cv2.findContours`
3. Bounding box	Obtains the bounding box (position and size in pixels) of each object	`cv2.boundingRect`
4. Scaling	Converts pixels to centimeters using the known card width	Rule of three: $\text{cm/pixel} = \frac{8{,}56}{w_{ref_px}}$
5. Annotation	Draws the boxes and displays the estimated measurements on the image	`cv2.rectangle` + `cv2.putText`

💡 **Real-world application:** this is exactly the technique used by *e-commerce* apps that estimate the size of a product from a photo taken next to a card, by agricultural systems that measure fruit on conveyor belts, and even by forensic experts who calculate the dimensions of traces at crime scenes — all based on the same idea: **a known ruler within the photo itself.**

9.6 Summary

This chapter, which concludes Part II of the book, presented:

- **Learned convolution and pooling:** the same mathematical convolution operation from Chapter 3, but with kernels treated as parameters adjusted through training, rather than defined manually;
- **Weight sharing and feature hierarchy** as properties that make CNNs efficient and capable of learning increasingly abstract representations in successive layers;
- **Training a CNN from scratch**, with performance comparable — and not necessarily superior — to the classical classifiers from Chapter 7 on a small and simple dataset, reinforcing that the choice of method should be proportional to the actual complexity of the problem;

- **Transfer learning**, experimentally demonstrated as an effective strategy for tasks with few labeled data, reusing a feature extractor already trained on a related task, and its limitations, evidenced by negative transfer between very distinct domains;
- **Large-scale applications** with pre-trained models for classification, detection (*Faster R-CNN*), and segmentation (*DeepLabV3*), following the same principle of transfer learning at an industrial scale;
- **Transfer learning applied to object detection**, fine-tuning a YOLO pre-trained on COCO to locate and classify synthetic geometric objects, illustrating the same principle of partial freezing in source and target domains that are even more distant from each other;
- **Semantic segmentation with U-Net**, implemented and trained from scratch on a synthetic low-contrast dataset, outperforming a classical thresholding baseline thanks to the skip connections between encoder and decoder;
- **Data engineering for Computer Vision with Roboflow**, using a dataset and a pre-trained model to perform inference in geometric object detection, as well as comparing datasets with different resolutions but with the same object classes;
- The integration of **computational geometry** (homography and calibration) and **deep learning** in two real applications that conclude the book: **augmented reality** and **photogrammetry**.

9.7 🤖 Using Gemini Notebook as a Complementary Tutor

In this edition, the use of **Gemini Notebook** is encouraged as a complementary learning tool. Based on artificial intelligence, the system uses exclusively the documents provided by the author as its source of knowledge, producing responses aligned with the content and approach adopted throughout this chapter.

🎓 Study with the Intelligent Tutor

🚀 ACCESS Gemini Notebook: CHAPTER 09

🌐 **Language and Programming Language**

The project for this chapter in Gemini Notebook was built using only the text in **Portuguese** and the code examples in **Python**. If you are studying from the English or French edition, or following the C++ track, the tutor's responses may not correspond exactly to the version you are reading.

⚠️ **Notice Regarding AI-Generated Content**

Although it is a valuable study support tool, Gemini Notebook may occasionally produce incomplete, inaccurate, or incorrect responses. It is recommended that you validate the information by consulting the chapter material, books, scientific articles, and other reliable academic sources. Whenever possible, run and experiment with the practical examples presented throughout the text to consolidate your understanding of the concepts.

9.8 Exercise List

The following exercises consolidate the concepts presented in this chapter through adaptations, experiments, and extensions of the algorithms developed throughout the text, using the `PyTorch`, `ultralytics`, and the didactic library `morph`.

1. **(10%)** Investigate the impact of depth on a convolutional architecture. Starting from the two-layer network of Practical Project 1, add a third convolutional layer with 32 filters before the fully connected layers. Train the new architecture keeping the same number of epochs and the same data split. Compare the accuracy on the test set and the total number of trainable parameters relative to the original

network, discussing whether the increase in depth brought a measurable benefit for images of dimension 8×8.

2. **(15%)** Evaluate the data threshold required in the target domain for training a CNN from scratch to become competitive with **transfer learning**. Varying the number of training samples available in domain B over $\{5, 10, 20, 40, 80\}$, measure the test accuracy for both strategies. Present the results in a line plot and determine from which data volume training from scratch achieves performance equivalent to the pre-trained extractor.

3. **(15%)** Investigate the **partial fine-tuning** strategy compared to total weight freezing. In the transfer learning scenario between digit domains, unfreeze the second convolutional layer (`conv2`) of the extractor so that it is updated together with the classification head during training on domain B. Compare the accuracy obtained with total freezing and with training from scratch, discussing the trade-off between adaptation capacity and the risk of overfitting.

4. **(20%)** Evaluate the influence of freeze depth on the performance of **YOLO** detectors subjected to transfer learning. Using the synthetic dataset of geometric shapes, perform fine-tuning while varying the backbone freezing parameter over $\{0, 5, 10, 15\}$. Record the mAP_{50} metric on the validation set for each configuration, present the data in a table, and discuss whether partial freezing is advantageous when the source (COCO) and target (geometric shapes) domains are significantly distinct.

5. **(20%)** Study the importance of **skip connections** in the **U-Net** architecture for semantic segmentation. Implement a variation `UNetSemAtalhos` that functions as a traditional convolutional *autoencoder*, removing the concatenations between the encoder and decoder stages. Train both models on the same base of synthetic nodules, compare the mean IoU on the validation set, and visually present the difference in the accuracy of the edges segmented by each method.

6. **(20%) — Challenge: semantic segmentation with *Roboflow*.** Use a *Roboflow* project of the *instance segmentation* type, containing the seven classes of geometric shapes used in this chapter. Export the *dataset* in the `coco-segmentation` format and develop a procedure to convert the polygons stored in the `_annotations.coco.json` files into multiclass semantic masks, in which each pixel receives the index of the corresponding class and the value `0` represents the background. Use the resulting images and masks to train `UNetCompacta`. Evaluate the mean IoU on the test set and visually compare the predicted masks with the original annotations. Discuss the main difficulties encountered in converting COCO annotations to masks and the effects of overlapping objects or objects belonging to different classes.

7. **(Bonus – 10%)** Develop an interactive system that combines **object detection (YOLO)** with **scale reference measurement (photogrammetry)**. Train the detector to identify two classes in a scene: a "Reference Card" (known dimension of 8,56 cm × 5,39 cm) and a "Target Object." When performing inference on a new image, use the pixel dimension of the detected card's *bounding box* to convert the target object's box dimensions into centimeters. Display the processed image with class labels, confidence probability, and estimated physical dimensions overlaid.

8. **(Bonus – 10%)** Develop a **stereo vision depth estimation** system from two images of the same scene obtained from different positions, simulating a pair of stereo cameras. Assume that the distance between the two capture positions (*baseline*) is known.

   Use one of the **object detection** methods presented in the chapter, such as YOLO, to locate the objects of interest in both images. For each detection, establish the correspondence between the same object in the two positions and determine its **disparity**. From the disparity, the *baseline*, and the camera parameters, use stereo geometry to estimate the distance of each object relative to the cameras.

   As an extension of the didactic library `morph`, modify the `showBoundBox` method so that, in addition to the **class** and **detection confidence**, it displays the **estimated distance of the object** over each *bounding box*. The result should allow visualizing, directly in the images, the class, the accuracy (confidence), and the depth of each detected object.

Present the two images with the detections, the correspondences between the objects, the disparity image, and a representation of the estimated depth. Discuss how the distance between the cameras, the accuracy of detection and correspondence, the image resolution, and the object's position in the scene influence the quality of the estimate.

For validation, use at least one object whose distance to the camera is known. Compare the estimated depth with the actual value and report the **absolute error** and the **relative error**. Also discuss the limitations of the method when an object is not correctly detected in both images or when the correspondence between the observed regions is ambiguous.

9.9 Closing of Part II

This chapter concludes Part II of the book and closes the sequence of content initiated in Chapter 6, dedicated to the representation, detection, description, and matching of features in images. Throughout these chapters, classical CV methods based on **handcrafted features**, such as Sobel, LBP, HOG, ORB, and Haar Cascade, were presented, as well as methods grounded in **automatically learned features**, represented by CNNs.

The examples and experiments developed show that none of these approaches is universally superior. The choice of the most appropriate technique depends on the characteristics of the problem, the availability of training data, the accuracy requirements, and the computational constraints of the application. In well-structured problems with limited data, classical descriptors often offer simple and efficient solutions. Conversely, more complex tasks tend to benefit from the learning capacity provided by CNNs.

Several study directions can deepen the concepts presented in this part of the book, among which the following stand out:

- **Modern CNN architectures**, such as ResNet, EfficientNet, and *Vision Transformers*, which expand representation capacity and performance in visual classification and recognition tasks;
- **Object detection and segmentation**, with emphasis on the YOLO families and on prompt-based segmentation models, such as *Segment Anything*;
- **Three-dimensional reconstruction and SLAM** (*Simultaneous Localization and Mapping*), which use multiple images to estimate scene geometry and the trajectory of moving cameras;
- **Generative image models**, such as generative adversarial networks (GANs) and diffusion models, capable of synthesizing realistic images from examples or textual descriptions.

The fundamentals developed throughout Part II constitute the basis for these and other advanced CV areas, in which the adequate representation of visual information remains a central element for the analysis and understanding of images.

Chapter References

The concepts and algorithms presented in this chapter were grounded in classic and contemporary references from the Deep Learning literature applied to VC:

- Mcculloch (1943), for proposing the first mathematical and logical abstraction of the artificial neuron, establishing the conceptual foundations of computational neural processing.
- Rosenblatt (1958), for the original formulation of the **Perceptron**, the precursor model of the artificial neuron used in modern deep learning architectures.
- Goodfellow (2016) and Lecun (2015), for the fundamentals of neural networks, convolution, activation functions, and training of deep models.
- Bishop (2006), for rigorous concepts of pattern recognition, probability, maximum likelihood estimation, and statistical methods applied to machine learning.
- Ronneberger (2015) for the **U-Net** architecture, used in semantic segmentation with skip connections between encoder and decoder.
- Redmon (2016), for the **YOLO** (*You Only Look Once*) architecture, used in object detection experiments with transfer learning.

- Ren (2015), for the **Faster R-CNN** architecture, employed as a pretrained object detection model.
- He (2016), for the **ResNet** architecture, the basis of several pretrained feature extractors used in this chapter.
- Chen (2018), for the **DeepLabV3** architecture, used as a pretrained semantic segmentation model.
- Kirillov (2023), for the **Segment Anything (SAM)** model, mentioned as a study direction for promptable segmentation.
- {google} (2025), referring to the **Gemini Notebook** tool, used in the preparation of the chapter's summary infographic and made available as supplementary study support.

Run Colab Open GitHub

9.10 💻 Practical Section with Programming Exercises

The present list of **Programming Exercises (PEs)** consolidates the theoretical formulations presented throughout Chapter 9 — Deep Learning for Computer Vision — through an applied practical track. Unlike the training of complete neural networks with PyTorch, which requires execution time and, at times, a GPU, the PEs in this chapter isolate the **intermediate quantities** of a real deep learning pipeline — the output of a single convolutional layer, the result of a pooling operation, the count of trainable parameters of an architecture, the overlap between candidate bounding boxes, the quality of a segmentation mask, and the non-maximum suppression filter — allowing each step of the reasoning to be manually validated without relying on machine learning libraries or actual training.

The sequencing of the exercises reproduces the conceptual flow of the chapter and increases in difficulty at each step: it begins with the manual calculation of the output of a **learned convolutional layer** (🟢), based on an already trained kernel and bias; it proceeds to the **pooling** operation (🟢, max and average), which reduces spatial resolution between convolutional blocks; it continues with the **count of trainable parameters** (🟡) of a complete CNN architecture, evidencing why weight sharing makes these networks so much more economical than an equivalent fully connected layer; it delves into the calculation of **Intersection over Union (IoU)** and **Non-Maximum Suppression (NMS)** (🟡), a post-processing step common to detectors such as Faster R-CNN and YOLO; it moves on to the **evaluation of segmentation masks** (🟠) using the same IoU and Dice metrics employed to compare the U-Net with the classic morphological baseline; and it concludes with an **integrated pipeline** (🔴), combining the output of an object detector (after NMS) with a real-world measurement by scale reference — the same principle of photogrammetry studied in the final integration of the chapter.

Whenever it makes sense, each exercise points to methods from the didactic library `morph.py` (the same one used throughout the chapter, imported as `mm`) that solve a step of the problem or serve as a reference to check your own calculations — without, however, replacing the reasoning that you must implement.

! Guidelines for Solving the Programming Exercises

In all exercises of this chapter, the discretization or numerical rounding steps must employ standard rounding to the nearest integer (*round half away from zero*), mitigating ambiguities in values with a fraction exactly equal to 0,5. Unless explicitly stated otherwise: (i) the "convolution" operation follows the convention adopted by deep learning frameworks — **cross-correlation**, without spatial inversion of the kernel, exactly as presented in the "Convolutional Layer" section; (ii) padding is performed with zeros; (iii) bounding boxes are specified in the corner-to-corner format (x_1, y_1, x_2, y_2), with $x_1 < x_2$ and $y_1 < y_2$; and (iv) vectors/matrices follow indexing starting from 0, with the `[row][column]` convention for two-dimensional structures.

🎯 Objective of this Notebook

This notebook allows you to develop, validate, organize, and test solutions for **Programming Exercises (EPs)** in interactive environments, such as Colab, using the same test cases as Moodle, and copying them there only when registering the official grade.

Download

Download `morph.py` and `testsuite.py` by running the cell below:

```
import os, urllib.request

url = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/morph/config.py"
if not os.path.exists("config.py"):
    urllib.request.urlretrieve(url, "config.py")

import config
config.setup(testsuite=True)
from morph import mm
from testsuite import TestSuite
```

```
Environment ready. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Running the Tests

To evaluate the tests, run `TestSuite("EP09_01.extension").run()` in a new cell, replacing the extension with that of the language used (`.py`, `.java`, `.c`, `.cpp`, `.js`, or `.r`). The system downloads the test cases from GitHub, runs the program, and calculates the grade automatically.

To test Python code directly, without saving a file, use `run_code(code)` by passing the code as a *string* in a variable named `code`:

```
code = """
# ... your code here ...
"""
TestSuite("EP09_01").run_code(code)
```

9.10.1 EP09_01 🟢 Manual 2D Convolution (Forward Pass of a Learned Layer)

PyTorch, presented in this chapter, executes `nn.Conv2d(x)` in a single call — but behind it is merely the cross-correlation between a (already trained) kernel and a neighborhood of the input, followed by the sum of a bias and an activation, exactly as formalized in the "Convolutional Layer" Section. The essential difference compared to the fixed-kernel convolution of Chapter 3 is that, here, the kernel and bias values **already come ready** (as if they had been learned by gradient), and it is up to you to manually reproduce the forward pass that the framework executes internally.

Before training a real CNN, you have been tasked with implementing this forward pass from scratch, for a single convolutional layer with a single input channel and a single output filter, including support for arbitrary padding and stride.

9.10.1.1 📋 Implementation Guidelines

1. **Input:** Read the dimensions $H \times W$ of the input feature map and, subsequently, its $H \times W$ real values.
2. **Kernel and bias:** Read the dimensions $k_h \times k_w$ of the (already trained) kernel, its real values, and the bias b (real, scalar).
3. **Hyperparameters:** Read the padding p (integer, number of zeros added to each border) and the stride s (integer, sliding step).

4. **Padding:** Add p zeros to each of the four borders of the input map before the correlation.
5. **Cross-correlation:** For each output position (i, j), compute

$$z(i,j) = b + \sum_{u=0}^{k_h-1} \sum_{v=0}^{k_w-1} K(u,v) \cdot X_{pad}(i \cdot s + u,\ j \cdot s + v),$$

   scanning the input **without** flipping the kernel (the convention of deep learning frameworks, different from classical mathematical convolution).
6. **Activation:** Apply ReLU to each value: $a(i,j) = \max(0, z(i,j))$.
7. **Output dimensions:** $O_h = \lfloor (H + 2p - k_h)/s \rfloor + 1$ and $O_w = \lfloor (W + 2p - k_w)/s \rfloor + 1$.
8. **Output:** Print O_h and O_w on the first line, followed by O_h lines with O_w real values each (the output feature map, already with ReLU applied), formatted with 4 decimal places.

9.10.1.2 📌 Computational Restrictions

- **One input channel, one output filter:** there is no need to handle multiple channels or multiple filters in this simplified version.
- **No kernel flipping:** implement cross-correlation, not the classical mathematical convolution with a flipped kernel — this is the operation that PyTorch (and most frameworks) calls "convolution".
- **Zero padding:** the p pixels added to each border are always 0.
- **Formatting:** all output values must have exactly 4 decimal places, even when the value is an integer (e.g., `2.0000`).

9.10.1.3 🧠 Theoretical Foundation

Element	Role in the convolutional layer
Kernel K	Parameters learned by gradient, analogous to the coefficients of a fixed filter from Chapter 3, but adjusted by backpropagation
Bias b	Learned offset, added after the correlation — allows the neuron to "fire" even with null input
Padding	Controls the spatial output dimension and prevents the loss of information at the borders at each layer
Stride	Controls the sliding step; values > 1 reduce spatial resolution, as a form of downsampling embedded within the convolution itself
ReLU	Introduces non-linearity after the linear combination, exactly as in the "Activation Function" Section

9.10.1.4 🧩 `morph.py` Methods That May Help

- `mm.readImg(h, w, dtype='float')` — directly reads an $h \times w$ matrix of real values from standard input, saving manual parsing of the feature map and the kernel.
- `mm.correlacao0(f, kernel, bias)` — implements the same cross-correlation sum + bias that you will compute by hand, but **without** support for padding or stride, and converts the result to `uint8` (truncating negative and decimal values). It may serve as a conceptual reference or to check the simplest case ($p = 0$, $s = 1$), but it does not replace your complete implementation — which must preserve sign, decimal places, padding, stride, and ReLU.

9.10.1.5 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integers H and W.
- Next H lines: W real values each (input map).
- Next line: Integers k_h and k_w.
- Next k_h lines: k_w real values each (kernel).
- Next line: Real b (bias).
- Next line: Integers p and s.

Output:

- Line 1: Integers O_h and O_w.
- Next O_h lines: O_w real values each, with 4 decimal places.

9.10.1.6 📌 Examples

Input	Output	Observation
3 3 1 2 0 0 1 2 1 0 1 2 2 1 1 1 1 -2 0 1	2 2 2.0000 3.0000 0.0000 2.0000	Padding 0, stride 1: 2×2 output without padding.
3 3 1 2 0 0 1 2 1 0 1 2 2 1 0 0 1 0 1 2	2 2 1.0000 0.0000 1.0000 2.0000	Padding 1, stride 2: input padded with zeros before the correlation.

```
%%writefile EP09_01.py
# Python code
```

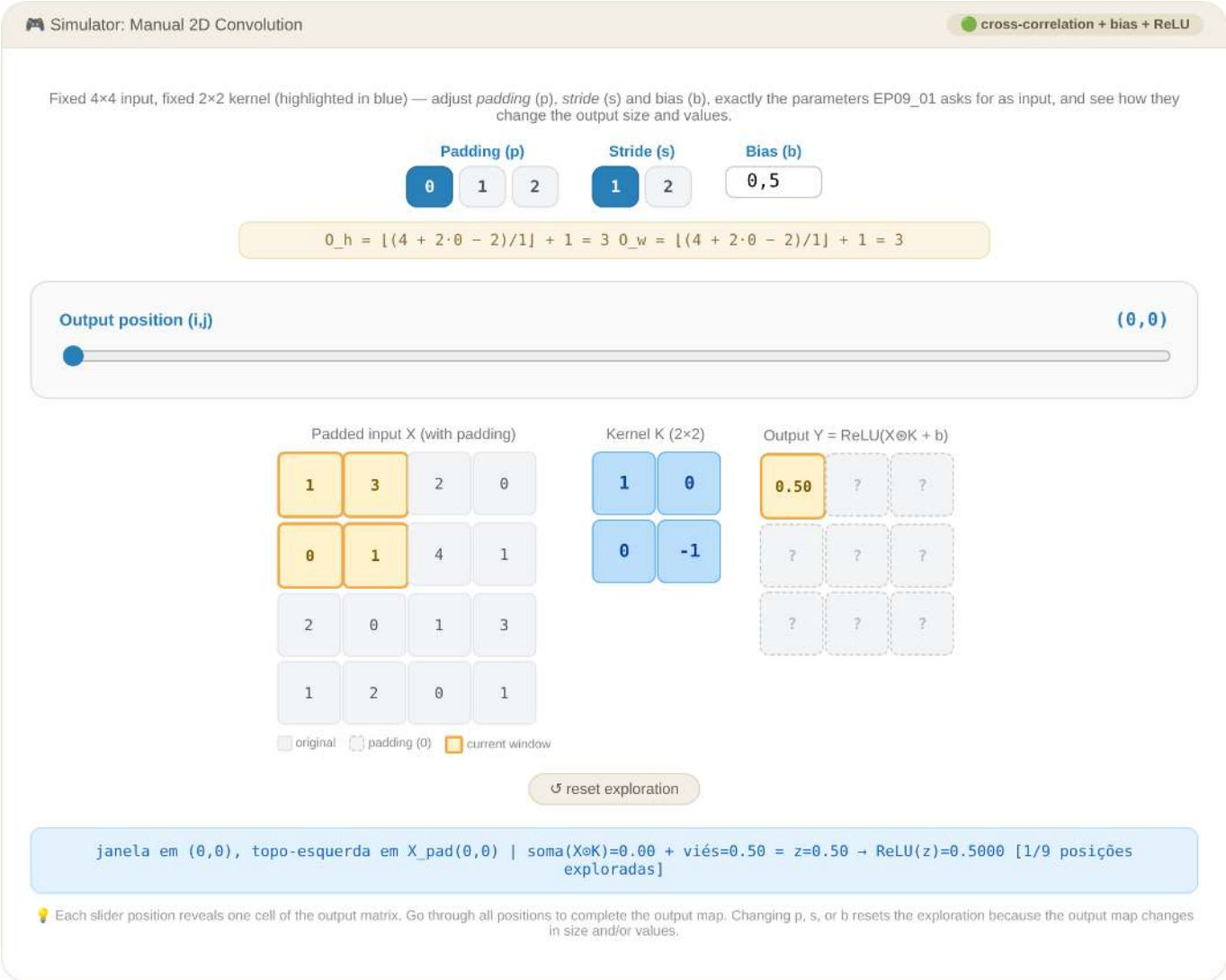
```
Overwriting EP09_01.py
```

```
TestSuite("EP09_01.py").run()
```

```
<IPython.core.display.HTML object>
```

9.10.2 EP09_02 🟢 Manual *Pooling* (Maximum and Average)

Between convolutional blocks, the typical architecture of a CNN interleaves ***pooling*** layers, which reduce the spatial resolution of the feature map without introducing new trainable parameters — unlike convolution, *pooling* has no weights: it simply summarizes each window of the input into a single value, either by a maximum or by an average, exactly as formalized in the Section "*Pooling*".



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-ep0901.html

Figure 9.43: EP09_01 Simulator: Manual 2D Convolution (cross-correlation + bias + ReLU, with adjustable *padding* and *stride*)

You have been tasked with implementing this operation using a square sliding window, without partial overlap at the borders (only complete windows), supporting the two most common types: `max` (preserves the most salient value, typically used to retain strong edges and textures) and `avg` (smooths the region, preserving average intensity information).

9.10.2.1 📋 Implementation Guidelines

1. **Input:** Read the dimensions $H \times W$ of the input feature map and its $H \times W$ real values.
2. **Window:** Read the integers k (size of the square window $k \times k$) and s (*stride*).
3. **Type:** Read a *string*, `max` or `avg`, indicating the type of *pooling*.
4. **No padding:** This operation **does not** use *padding*; windows that would exceed the input border are discarded.
5. **Calculation:** For each output position (i, j), compute the maximum or the average of the $k \times k$ values of the corresponding window, starting at $(i \cdot s,\ j \cdot s)$.
6. **Output dimensions:** $O_h = \lfloor (H - k)/s \rfloor + 1$ and $O_w = \lfloor (W - k)/s \rfloor + 1$.
7. **Output:** Print O_h and O_w on the first line, followed by O_h lines with O_w real values each, formatted with 4 decimal places.

9.10.2.2 📌 Computational Constraints

- **Square window:** $k \times k$, with no support for rectangular windows in this version.
- **No *padding*:** only windows fully contained within the input are considered — dimensions that "remain" are simply discarded.
- **`avg` uses real division:** the average is always sum/k^2, even when the result has many decimal places — round only in the final formatting, according to the general guideline of the chapter.
- **Formatting:** all output values with exactly 4 decimal places.

9.10.2.3 🧠 Theoretical Foundation

Element	Role in the architecture
Maximum *pooling*	Preserves the strongest activation of the window; common after convolutional layers to retain salient edges and textures
Average *pooling*	Smooths the region, preserving average intensity; common in final layers (*global average pooling*)
Absence of parameters	Differentiates *pooling* from convolution: reduces spatial resolution without additional training cost
Resolution reduction	Contributes to invariance to small translations and to the reduction of the computational cost of subsequent layers

9.10.2.4 🧩 Methods from `morph.py` that may help

`morph.py` does not implement *pooling* with subsampling directly, but two families of operations show the same idea from another perspective, useful for checking your intuition:

- `mm.dil(f, Bc)` / `mm.dil0(f, B)` — morphological dilation: replaces each pixel by the **maximum** of its neighborhood defined by the structuring element B (e.g., `mm.sebox(n)` for a $(2n+1) \times (2n+1)$ window). It is, conceptually, a "*max-pooling* without subsampling" (produces an image of the same size, rather than a reduced one).
- `mm.blur(f, N)` — averaging smoothing in an $N \times N$ window, analogous to *avg-pooling*, also without resolution reduction.
- `mm.readImg(h, w, dtype='float')` — useful for reading the input map in floating point.

9.10.2.5 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integers H and W.
- Next H lines: W real values each.
- Next line: Integers k and s.
- Next line: `max` or `avg`.

Output:

- Line 1: Integers O_h and O_w.
- Next O_h lines: O_w real values each, with 4 decimal places.

9.10.2.6 📌 Examples

Input	Output	Observation
4 4 1 3 2 4 5 6 1 2 2 1 0 3 4 2 5 1 2 2 max	2 2 6.0000 4.0000 4.0000 5.0000	Maximum *pooling*, 2×2 window, *stride* 2.
4 4 1 3 2 4 5 6 1 2 2 1 0 3 4 2 5 1 2 2 avg	2 2 3.7500 2.2500 2.2500 2.2500	Average *pooling* over the same windows.

```
%%writefile EP09_02.py
# Python code
```

```
Overwriting EP09_02.py
```

```
TestSuite("EP09_02.py").run()
```

```
<IPython.core.display.HTML object>
```

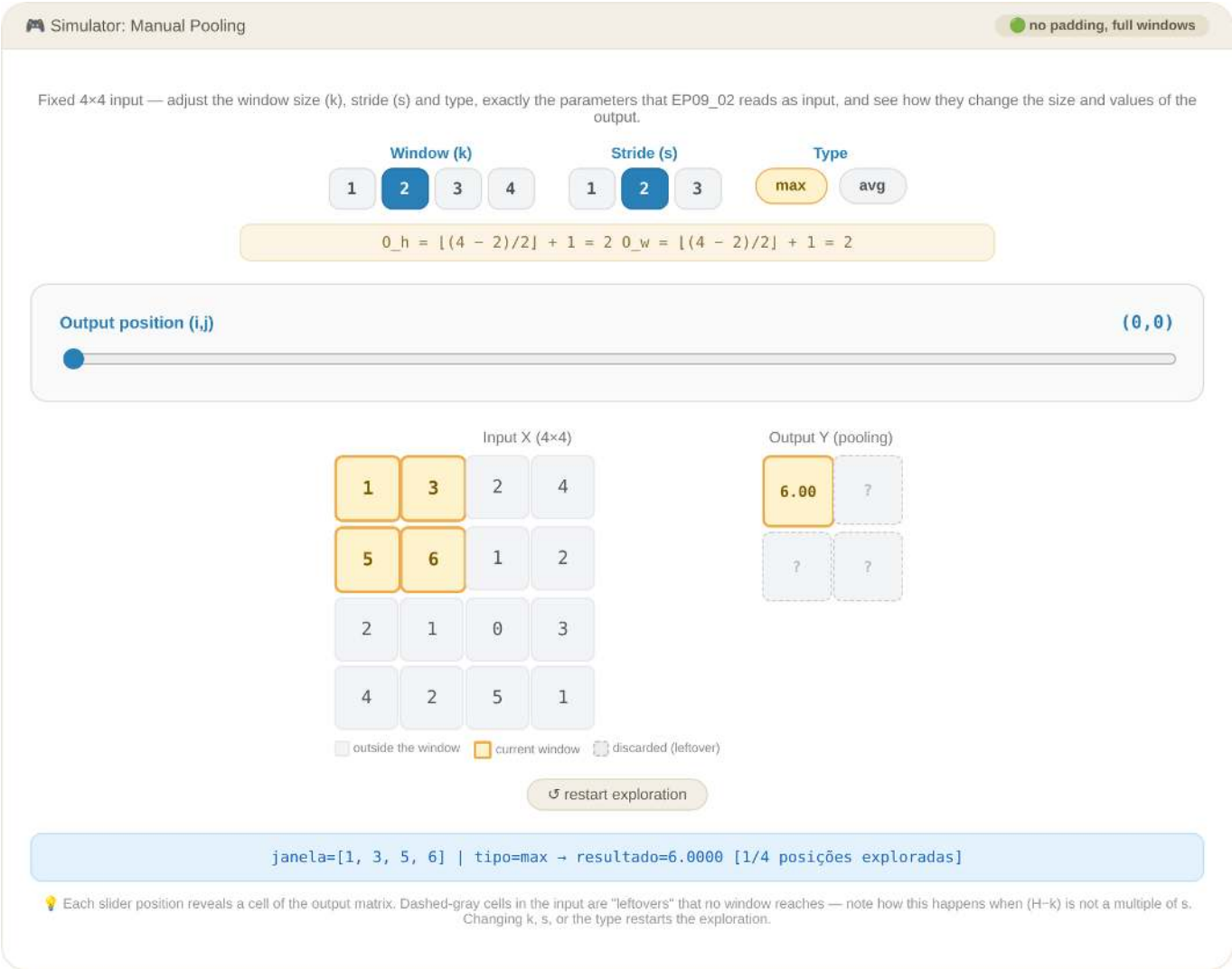
9.10.3 EP09_03 🟡 Counting Trainable Parameters of a CNN

This EP formalizes the counting of trainable parameters of a *CNN*. Given the textual description of a small architecture, composed of convolutional, pooling, and fully connected layers, determine, for each layer, the number of trainable parameters and the network total.

The architecture must be interpreted **sequentially**: the output of a convolutional layer becomes the input of the next compatible layer. Thus, the number of channels produced by a `CONV` layer determines the number of input channels (`cin`) of the following convolutional layer.

In a convolutional layer, it is important to distinguish **input channels** and **output channels**:

- c_{in} (*channels in*) is the number of **channels entering the layer**. A grayscale image has $c_{in} = 1$, while an RGB image has $c_{in} = 3$. In an intermediate convolutional layer, `cin` is typically equal to the number of channels produced by the preceding `CONV` layer.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-ep0902.html

Figure 9.44: Simulator EP09_02: Manual Pooling (max vs. mean, with adjustable window k and stride s)

- c_{out} (*channels out*) is the number of **channels produced by the layer**. It equals the number of filters used. Therefore, if a layer has 16 filters, it produces $c_{out} = 16$ channels.

For example, consider the sequence:

```
CONV 3 3 1 8 1
POOL
CONV 3 3 8 16 1
POOL
FC 784 10 1
```

The first convolution receives an image with one channel and produces 8 channels. After pooling, the second convolution receives those 8 channels and produces 16 channels. The `POOL` layer does not alter the number of channels; it may only reduce the spatial dimensions. The `FC` layer receives the number of inputs specified in its own description.

Each convolutional filter has dimensions

$$k_h \times k_w \times c_{in}.$$

Thus, a layer with c_{out} filters has

$$k_h \cdot k_w \cdot c_{in} \cdot c_{out}$$

weights. If there is a bias, one parameter is added for each filter, totaling an additional c_{out} parameters.

The central point of this exercise is to observe that the number of parameters in a convolutional layer **does not depend on the spatial dimensions** $(H \times W)$ of the feature map. This occurs due to **weight sharing**: the same filter is reused at different positions of the input.

9.10.3.1 📋 Implementation Guidelines

1. **Input:** Read the integer L (number of layers in the architecture, in the order they are applied).
2. **Layers:** Read L lines, each describing a layer in one of three formats:
   - `CONV kh kw cin cout bias` — convolutional layer with *kernel* $k_h \times k_w$, c_{in} input channels, c_{out} output channels, and `bias` (0 or 1), indicating whether there is a bias per filter;
   - `POOL` — pooling layer (max or average), which has no trainable parameters and preserves the number of channels;
   - `FC in out bias` — fully connected layer with `in` inputs, `out` outputs, and `bias` (0 or 1), indicating whether there is a bias per neuron.
3. **Consistency between `CONV` layers:** in a sequence of convolutional layers, the `cin` of a layer must correspond to the `cout` of the preceding convolutional layer. A `POOL` layer does not alter this number of channels.

   For example:

   ```
   CONV 3 3 1 8 1
   POOL
   CONV 3 3 8 16 1
   ```

   The first `CONV` produces 8 channels, which are received by the second `CONV`. Therefore, in the second layer, `cin=8` and `cout=16`.
4. **Parameters of a `CONV` layer:**

   Each of the c_{out} filters has $k_h \cdot k_w \cdot c_{in}$ weights. Therefore,

$$P_{\text{CONV}} = k_h \cdot k_w \cdot c_{in} \cdot c_{out} + c_{out} \cdot \text{bias}.$$

5. **Parameters of an `FC` layer:**

$$P_{\text{FC}} = \text{in} \cdot \text{out} + \text{out} \cdot \text{bias}.$$

6. **Parameters of a `POOL` layer:** always 0.
7. **Network total:** sum the trainable parameters of all layers.
8. **Output:** For each layer, in reading order, print `Camada i: P`, where i starts at 1 and P is the number of parameters for that layer. At the end, print `Total: T`.

9.10.3.2 📐 Example to understand `cin` and `cout`

Consider the sequence:

```
CONV 3 3 1 8 1
POOL
CONV 3 3 8 16 1
```

In the first layer:

- `cin=1`: one channel enters;
- `cout=8`: there are 8 filters and, therefore, 8 channels exit.

Each filter has

$$3 \cdot 3 \cdot 1 = 9$$

weights. Since there are 8 filters:

$$9 \cdot 8 = 72$$

weights. With one bias per filter:

$$72 + 8 = 80.$$

In the second layer:

- `cin=8`: the 8 channels produced by the first `CONV` enter;
- `cout=16`: there are 16 filters and, therefore, 16 channels exit.

Each filter has

$$3 \cdot 3 \cdot 8 = 72$$

weights. Since there are 16 filters:

$$72 \cdot 16 = 1152$$

weights. With 16 biases:

$$1152 + 16 = 1168.$$

Thus, the two layers have, respectively, **80** and **1168 trainable parameters**.

Note that `cout` **is not** *cin* multiplied by the number of filters. The number of filters is exactly `cout`: each filter combines all input channels and produces **a single output channel**.

9.10.3.3 📌 Computational Constraints

- **Independence from spatial dimension:** the input does not provide $H \times W$. The count for a `CONV` layer depends only on `kh`, `kw`, `cin`, and `cout`.
- **Channel consistency:** for two consecutive `CONV` layers, the `cin` of the second must equal the `cout` of the first. A `POOL` layer preserves the number of channels.
- **`bias` always 0 or 1:** multiply the bias term directly by this value.
- **`POOL` layers with no additional arguments:** the line contains only the word `POOL`.
- **`FC` layers:** the number of inputs `in` is provided explicitly. It is not necessary to compute the spatial dimensions produced by previous layers.
- All numerical input values are non-negative integers.

9.10.3.4 🧠 Theoretical Foundation

Element	Role in parameter counting
c_{in}	Number of channels received by the layer
c_{out}	Number of filters and, therefore, of channels produced by the layer
Convolutional filter	Each filter has $k_h \cdot k_w \cdot c_{in}$ weights and produces one output channel
Weight sharing	The same filter is reused at different input positions, making the count independent of $H \times W$
Bias	A single additional parameter per filter (`CONV`) or per neuron (`FC`)
Pooling	May alter $H \times W$, but has no trainable parameters and preserves the number of channels
`FC` layer	Has one weight for each combination of input and output neuron

9.10.3.5 🧩 `morph.py` methods that may help

This exercise is purely arithmetic and does not directly use functions from `morph.py`. The count can, however, be checked in a real architecture implemented in PyTorch using:

```
sum(p.numel() for p in modelo.parameters())
```

This expression counts the model parameters, including weights and biases.

9.10.3.6 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integer L.
- Next L lines: description of each layer, in the format `CONV kh kw cin cout bias`, `POOL`, or `FC in out bias`.

Output:

- L lines in the format `Camada i: P`.
- Last line: `Total: T`.

9.10.3.7 📌 Examples

Input	Output	Observation
3 CONV 3 3 1 8 1 POOL FC 1352 10 1	Camada 1: 80 Camada 2: 0 Camada 3: 13530 Total: 13610	Simple network with one convolution, *pooling*, and classification layer.
5 CONV 3 3 1 8 1 POOL CONV 3 3 8 16 1 POOL FC 400 10 1	Camada 1: 80 Camada 2: 0 Camada 3: 1168 Camada 4: 0 Camada 5: 4010 Total: 5258	Small CNN with two convolutions, two *poolings*, and one fully connected layer.
6 CONV 3 3 1 8 1 POOL CONV 3 3 8 16 1 POOL FC 256 32 1 FC 32 10 1	Camada 1: 80 Camada 2: 0 Camada 3: 1168 Camada 4: 0 Camada 5: 8224 Camada 6: 330 Total: 9802	Small CNN with two convolutions, intermediate *pooling*, and two fully connected layers for classification.

```
%%writefile EP09_03.py
# Python code
```

```
Overwriting EP09_03.py
```

```
TestSuite("EP09_03.py").run()
```

```
<IPython.core.display.HTML object>
```

9.10.4 EP09_04 🟡 Intersection over Union (IoU) and Non-Maximum Suppression (NMS)

Object detection models can produce **multiple candidate bounding boxes** for the same object, with different positions and confidence scores. The post-processing step responsible for eliminating these redundant detections is **Non-Maximum Suppression (NMS)**, whose fundamental operation uses the **Intersection over Union (IoU)** metric.

NMS uses this measure to decide which boxes should be kept. In general, the box with the highest confidence is selected first; then, boxes that have an IoU above a given threshold with the selected box are considered redundant and removed. The process is repeated until no candidate boxes remain.

In this exercise, you must implement the NMS algorithm from scratch, calculating the IoU between boxes and successively applying the selection and suppression criteria to produce the final set of detections.

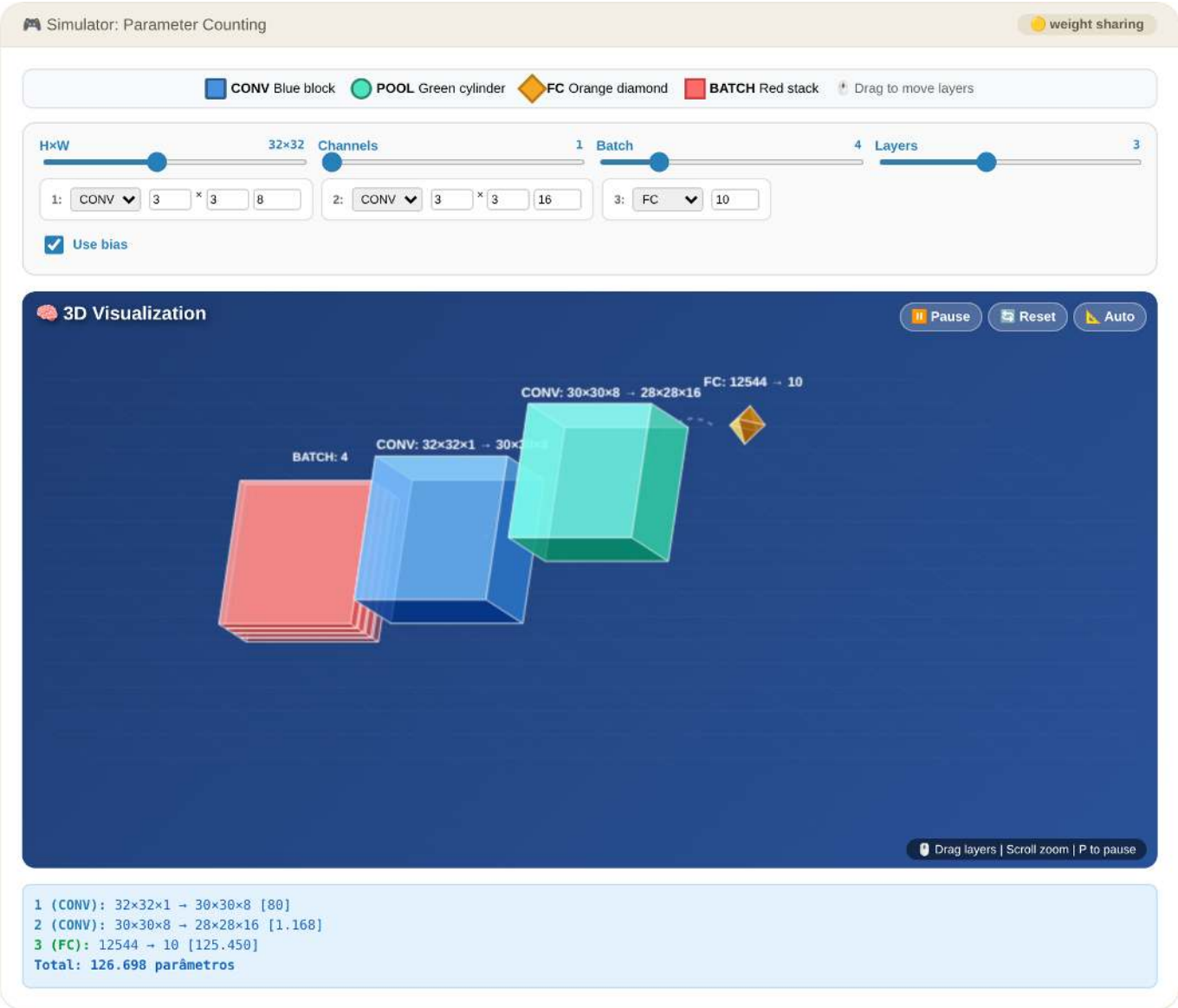
9.10.4.1 📋 Implementation Guidelines

1. **Input:** Read the integer N (number of candidate boxes) and the real threshold τ (IoU threshold for suppression), on the same line.

2. **Boxes:** Read N lines, each with five real values:

   `x1 y1 x2 y2 score`

   where (x_1, y_1) represents the top-left corner, (x_2, y_2) the bottom-right corner, and `score` the confidence score.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-ep0903.html

Figure 9.45: Simulator EP09_03: Parameter Counting — Convolution vs. Fully Connected Layer

3. **Intersection over Union:** For two boxes A and B,

$$IoU(A, B) = \frac{\text{Area}(A \cap B)}{\text{Area}(A \cup B)}.$$

The intersection area must be computed from the overlap of the intervals in x and y. If there is no overlap, the intersection area is zero.

4. **Greedy NMS algorithm:**

   a. Sort the boxes by `score` in descending order. In case of ties, maintain the original reading order.

   b. Select the box with the highest score among the remaining boxes and add it to the output set.

   c. Compute the IoU between the selected box and **all remaining boxes**. Suppress those boxes for which

$$\text{IoU} > \tau.$$

   d. Repeat steps (b) and (c) until no boxes remain.

5. **Output:** For each kept box, in the order it was selected, print its original index (reading position, starting at 0) and its `score`, formatted with 4 decimal places. At the end, print:

```
Total kept: X
```

9.10.4.2 📌 Computational Constraints

- **Strict suppression:** only boxes with IoU $> \tau$ are suppressed. Boxes with IoU $= \tau$ are kept.
- **Original indices:** the output refers to the position in which each box was read from the input (starting at 0), not its position after sorting.
- **Stable sorting:** in case of equal `score` values, the original reading order must be preserved.
- **Axis-aligned rectangles:** all boxes are specified by two corners, with $x_1 < x_2$ and $y_1 < y_2$ guaranteed in the input.
- **Coordinates and scores:** real values may be positive or negative, according to the limits defined by the input, but the box dimensions are always positive.

9.10.4.3 🧠 Theoretical Background

Element	Role in detection post-processing
IoU	Quantifies the spatial overlap between two boxes; IoU $= 1$ for identical boxes and IoU $= 0$ for boxes with no overlap
Confidence-based sorting	Ensures that the box with the highest `score` is analyzed first
Threshold τ	Defines the amount of overlap required for a box to be considered redundant
Suppression	Removes boxes that exhibit large overlap with an already selected box
Distant boxes	Have IoU close to zero and, in general, are not suppressed by this rule

9.10.4.4 🧩 Methods from `morph.py` that may help

- `mm.IoU(boxA, boxB)` — computes the IoU metric, but expects boxes in the format (x, y, w, h), i.e., top-left corner, width, and height. The input of this exercise uses the format (x_1, y_1, x_2, y_2). The conversion is straightforward:

$$w = x_2 - x_1, \qquad h = y_2 - y_1.$$

Using this function is optional. The main goal of the exercise is to correctly implement the NMS selection and suppression process.

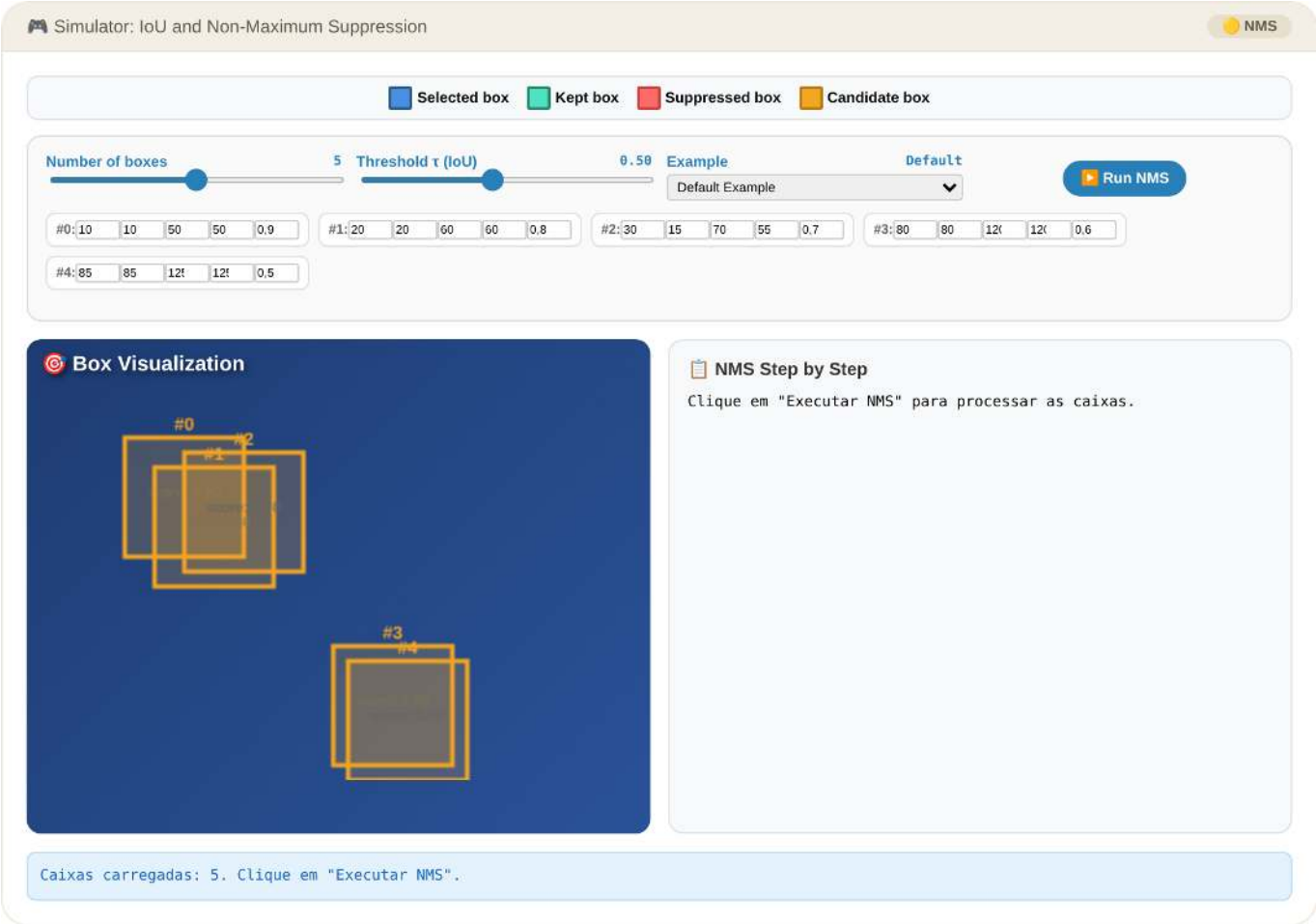
9.10.4.5 📦 Input and Output Specification (VPL)

Input:

- Line 1: integer N and real τ.
- Next N lines: x_1 y_1 x_2 y_2 score.

Output:

- One line per kept box, in selection order: `index score`.
- Last line: `Total kept: X`.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-ep0904.html

Figure 9.46: EP09_04 Simulator: IoU and Non-Maximum Suppression (NMS)

```
%%writefile EP09_04.py
# Python code
```

```
Overwriting EP09_04.py
```

```
TestSuite("EP09_04.py").run()
```

```
<IPython.core.display.HTML object>
```

9.10.5 EP09_05 🟠 Segmentation Evaluation: Pixel-wise IoU and Dice

Block 2 of the "Semantic Segmentation with U-Net Architecture" section defines, in a few lines, the `iou_mascaras` function, used to measure the quality of the classic morphological baseline (smoothing + Otsu + opening) and, later, of the trained U-Net itself. Unlike the IoU from EP09_04 — computed over **bounding boxes** (rectangular regions described by four numbers) —, the segmentation IoU is computed **pixel by pixel**: each image position is compared individually between the predicted mask and the reference mask.

You have been tasked with generalizing this evaluation, implementing not only the pixel-wise IoU but also the **Dice coefficient**, another overlap metric widely used in medical segmentation (including in the `perda_dice` function, mentioned in the same block of the chapter as the basis for the loss function used to train the U-Net).

9.10.5.1 📋 Implementation Guidelines

1. **Input:** Read the dimensions $H \times W$ of the masks.
2. **Predicted mask:** Read H lines with W integer values (0 or 1) each — for example, the output of a U-Net after thresholding at 0.5 over the sigmoid, as in Block 4 of the chapter.
3. **Reference mask:** Read another H lines with W integer values (0 or 1) each — the *ground truth*.
4. **Intersection and union:** Considering each pixel as belonging to the object when its value is different from zero,
$$\text{intersection} = \sum_{i,j} \mathbb{1}[P_{ij} = 1 \wedge R_{ij} = 1], \qquad \text{union} = \sum_{i,j} \mathbb{1}[P_{ij} = 1 \vee R_{ij} = 1].$$
5. **Pixel-wise IoU:**
$$\text{IoU} = \frac{\text{intersection}}{\text{union}}.$$
6. **Dice coefficient:**
$$\text{Dice} = \frac{2 \cdot \text{intersection}}{|P| + |R|},$$
where $|P|$ and $|R|$ are the total number of object pixels in each mask.
7. **Convention for empty masks:** if **both** masks have no object pixels (union $= 0$ and $|P| + |R| = 0$), consider the correspondence trivially perfect: IoU = Dice = 1.0.
8. **Output:** Two lines, `IoU: X.XXXX` and `Dice: X.XXXX`, each value with 4 decimal places.

9.10.5.2 📌 Computational Constraints

- **Any non-zero value counts as an object:** treat values different from 0 (not just 1) as belonging to the mask, replicating the `predita > 0` check used in `iou_mascaras` in the chapter.
- **Same dimensions:** the two masks always have exactly $H \times W$ elements.
- **Empty convention:** apply the rule from item 7 **only** when both masks are completely empty; if only one is empty, the intersection is 0 and the resulting IoU/Dice will also be 0.

9.10.5.3 🧠 Theoretical Foundation

Element	Role in segmentation evaluation
Pixel-wise IoU	Generalizes the metric from EP09_04 to arbitrarily shaped regions — not just rectangles — by comparing predicted and reference masks position by position
Dice coefficient	Metric related to IoU (always Dice $\geq$ IoU), more sensitive to small intersections and widely used as a loss function in segmentation (`perda_dice` function from the chapter)
Empty mask convention	Avoids division by zero and recognizes that "no predicted object, no real object" is, by definition, a correct match
Classic vs. U-Net comparison	The chapter uses exactly this type of metric to numerically justify why the U-Net surpasses the morphological baseline in low-contrast scenarios

9.10.5.4 🧩 Methods from `morph.py` that may help

- `mm.readImg(h, w, dtype='uint8')` — directly reads each binary mask $h \times w$ from standard input (the values 0/1 fit perfectly in the standard integer type).
- The `iou_mascaras` function itself, defined in Block 2 of the U-Net section of the chapter (not part of `morph.py`, but of the chapter's code), is the direct inspiration for this exercise — it is worth re-reading those few lines before coding.
- For an optional extension (not required by this EP), `mm.connectedComponents` or `mm.label0` (seen in the context of connected component analysis) would allow labeling each nodule individually and computing the IoU **per component**, instead of over the entire mask.

9.10.5.5 📦 Input and Output Specification (VPL)

Input:

- Line 1: Integers H and W.
- Next H lines: W integer values (0 or 1) — predicted mask.
- Next H lines: W integer values (0 or 1) — reference mask.

Output:

- Line 1: `IoU: X.XXXX`.
- Line 2: `Dice: X.XXXX`.

💡 Illustrative Example

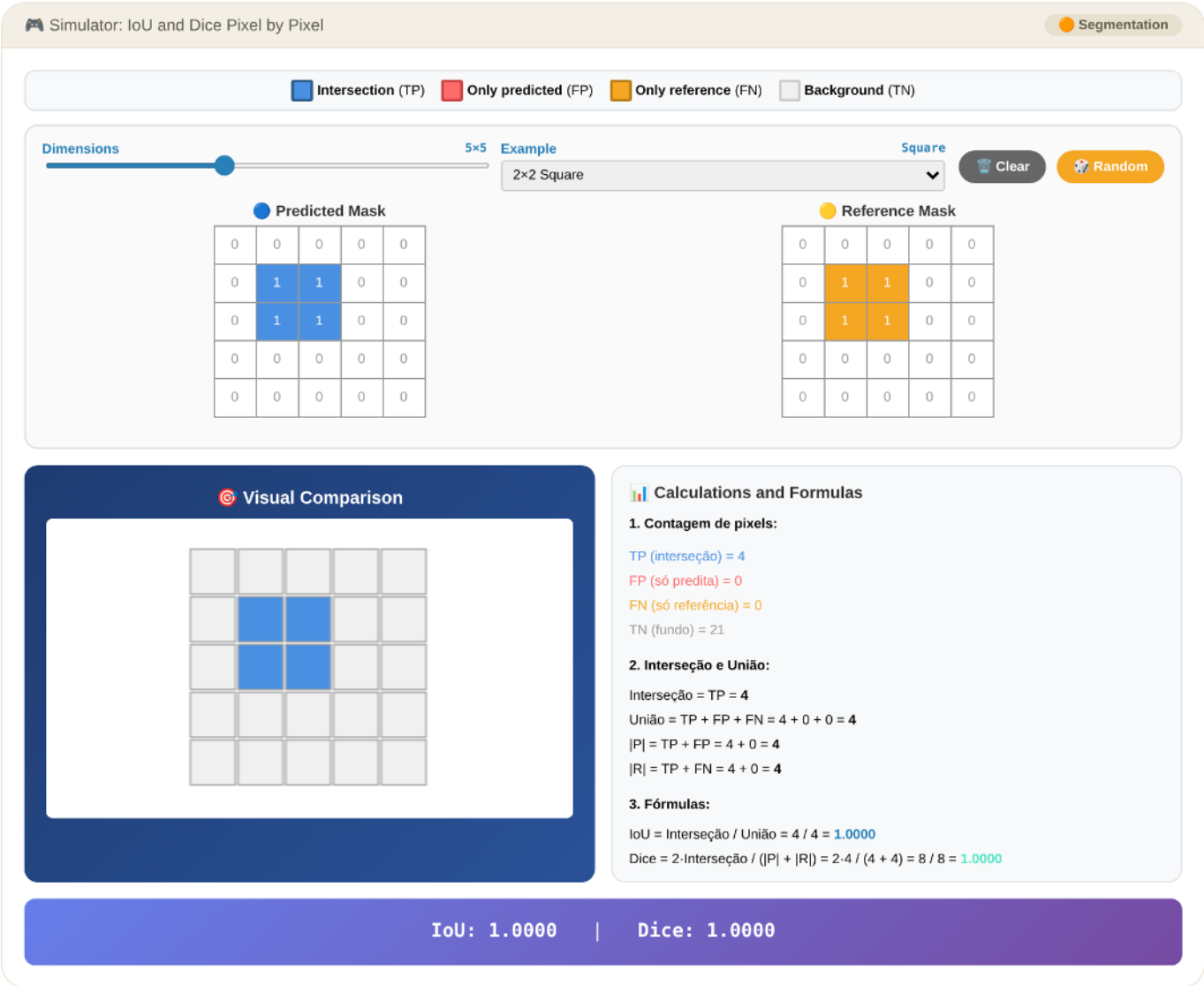
Consider a predicted mask with a 2×2 square of active pixels and a reference shifted by one column, overlapping on only half of the area:

```
Predicted       Reference
0 0 0 0         0 0 0 0
0 1 1 0         0 0 1 1
0 1 1 0         0 0 1 1
0 0 0 0         0 0 0 0
```

Intersection = 2 pixels, union = 6 pixels $(4 + 4 - 2)$, therefore IoU $= 2/6 \approx 0.3333$ and Dice $= 2 \cdot 2/(4 + 4) = 0.5000$ — note that Dice is always equal to or greater than IoU for the same overlap.

9.10.5.6 📌 Examples

Input	Output	Observation
4 4 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0	IoU: 0.3333 Dice: 0.5000	4×4 masks with partial overlap of 2 pixels.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-ep0905.html

Figure 9.47: Simulator EP09_05: Segmentation Evaluation — IoU and Dice Pixel by Pixel

```
%%writefile EP09_05.py
# Python code
```

```
Overwriting EP09_05.py
```

```
TestSuite("EP09_05.py").run()
```

```
<IPython.core.display.HTML object>
```

9.10.6 EP09_06 🔴 Integrated Pipeline: From Detection to Real-World Measurement

This final exercise integrates the two detection exercises and the principle of **photogrammetry** presented in the section "Photogrammetry and Scale Reference" — exactly the same calculation implemented in the scale-reference measurement figure in this chapter. The scenario reproduces a realistic situation: a detector (Faster R-CNN or YOLO) generates **several overlapping candidate boxes** for the same object of interest; after filtering them by NMS, the surviving box with the highest confidence is used, together with a reference box of known real width (such as the 8.56 cm card), to estimate the real dimensions of the detected object.

9.10.6.1 📋 Implementation Guidelines

1. **Known reference:** Read the real value L_{ref} (real width of the reference object, in cm) and then the four real values x_1 y_1 x_2 y_2 of its bounding box in pixels (already known, with no need for detection).
2. **Candidates of the object to measure:** Read the integer N (number of candidate boxes produced by the detector for the object of interest) and the real threshold τ; then read the N lines of candidate boxes, each with x_1 y_1 x_2 y_2 score.
3. **Step 1 — NMS:** Apply exactly the Non-Maximum Suppression algorithm from EP09_04 to the N candidate boxes, using the threshold τ, to eliminate redundant detections of the same object.
4. **Step 2 — Final box selection:** After NMS, the box with the highest `score` among those retained is the final detection of the object (the input guarantees that all candidate boxes correspond to a single physical object, so the first box selected by NMS is already the final result).
5. **Step 3 — Scale-reference measurement:** Calculate the ratio cm/pixel $= L_{ref}$/reference width in pixels and apply it to both the width and height (in pixels) of the final object box, obtaining its estimated real dimensions in centimeters.
6. **Output:** First, one line per box retained after NMS (same format as EP09_04): `index score`. Then the line `Total kept: X`. Finally, the line `Object: L x A cm`, where L and A are the estimated width and height of the object, each with 2 decimal places.

9.10.6.2 📌 Computational Constraints

- **Reuse the NMS from EP09_04** in full — same tie-breaking rule, same suppression criterion (IoU $> \tau$).
- **The reference does not undergo NMS:** its box is given directly, without competing candidates.
- **Single ratio for width and height:** just as in the photogrammetry figure in the chapter, the same cm/pixel ratio (derived from the reference width) is applied to both the width and height of the object — there is no separate vertical calibration.

9.10.6.3 🧠 Theoretical Foundation

Step	Chapter concept
Multiple candidate boxes	Raw output of a detector such as Faster R-CNN or YOLO, before post-processing
NMS (EP09_04)	Filters redundant detections, keeping only the most confident one for the object
Known scale reference	Same principle as the 8.56 cm card used in the section "Photogrammetry and Scale Reference"
Pixel → centimeter conversion	Simple rule of three: cm/pixel $= L_{ref}/w_{ref_px}$, applied to the final object box

9.10.6.4 🧩 `morph.py` Methods That May Help

- `mm.IoU(boxA, boxB)` — the same function suggested in EP09_04, here reused within the NMS step of this integrated *pipeline* (remember the format conversion: $w = x_2 - x_1$, $h = y_2 - y_1$).
- If you have already solved EP09_04 by encapsulating NMS in your own function, this is the ideal time to **reuse that code** — integrating modules that have already been tested individually is exactly the engineering practice this exercise aims to reinforce.

9.10.6.5 📦 Input and Output Specification (VPL)

Input:

- Line 1: Real L_{ref}.
- Line 2: x_1 y_1 x_2 y_2 of the reference box.
- Line 3: Integer N and real τ.
- Next N lines: x_1 y_1 x_2 y_2 score of the candidate boxes for the object.

Output:

- One line per box retained after NMS: `index score`.
- Next line: `Total kept: X`.
- Last line: `Object: L x A cm`.

9.10.6.6 📌 Examples

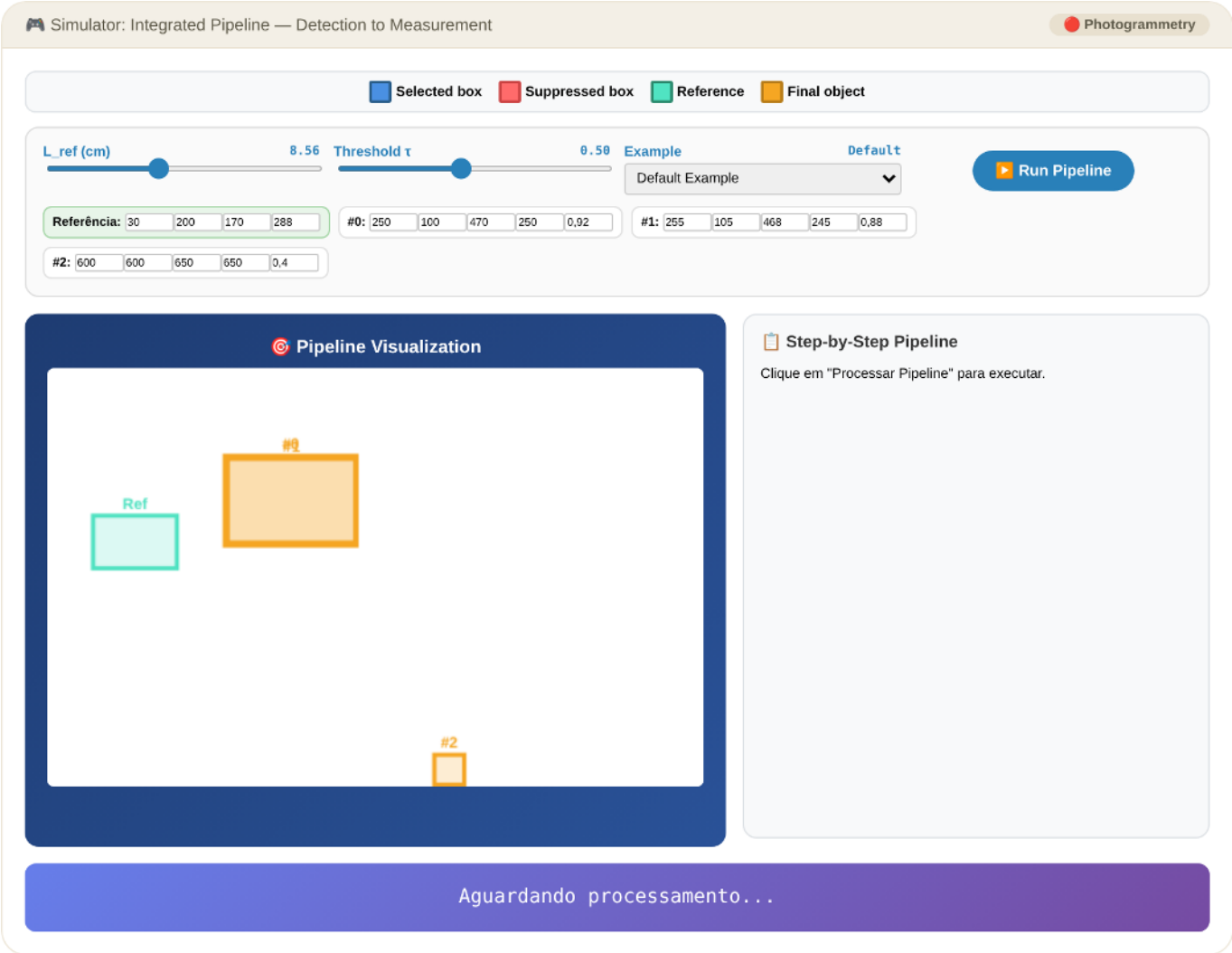
Input	Output	Observation
8.56 30 200 170 288 3 0.5 250 100 470 250 0.92 255 105 468 245 0.88 600 600 650 650 0.40	0 0.9200 2 0.4000 Total kept: 2 Object: 13.45 x 9.17 cm	Box 1 is suppressed because it strongly overlaps box 0; the final detection of the object is box 0.

```
%%writefile EP09_06.py
# Python code
```

```
Overwriting EP09_06.py
```

```
TestSuite("EP09_06.py").run()
```

```
<IPython.core.display.HTML object>
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.en/cap09/sim-ep0906.html

Figure 9.48: EP09_06 Simulator: Integrated Pipeline — From Real-World Detection to Measurement

References

BAHDANAU, Dzmitry; CHO, Kyunghyun; BENGIO, Yoshua. Neural machine translation by jointly learning to align and translate. *In*: 2015. Disponível em: <https://arxiv.org/abs/1409.0473>

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.

DOUGHERTY, Edward R.; LOTUFO, Roberto A. **Hands-on morphological image processing**. *[S.l.]*: SPIE press, 2003. v. 59

KNUTH, Donald E. Literate programming. **The Computer Journal**, v. 27, n. 2, p. 97–111, 1984.

LOTUFO, Roberto A. *et al.* MMachLib functions and MMach operators. *In*: 1997.

MATHERON, Georges. **Random sets and integral geometry**. New York: John Wiley & Sons, 1975.

VIOLA, Paul; JONES, Michael. Rapid object detection using a boosted cascade of simple features. *In*: IEEE Computer Society, 2001.

ZAMPIROLLI, Francisco A. **MCTest: Como criar e corrigir exames parametrizados**. *[S.l.]*: Independente, 2023.

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.

ZAMPIROLLI, Francisco de Assis *et al.* Intelligent feedback for individualized introductory programming exercises. **Computer Applications in Engineering Education**, v. 34, n. 1, p. e70132, 2026.

Appendix A

About MCTest

MCTest is an open-source system for creating and automatically grading parameterized exams (Zampirolli, 2023). It supports multiple-choice, essay, and programming questions, the latter integrated with Moodle's VPL, described in Appendix B.

The version of MCTest used in this book is **5.4**, available at https://github.com/fzampirolli/mctest. The repository's `book` subfolder contains versions **5.3**, described in (Zampirolli, 2023), and **5.4**, used to generate the exams presented in this book and currently in production at UFABC (https://mctest.ufabc.edu.br).

A.1 Installing MCTest

The recommended installation uses VirtualBox with Ubuntu 22.04. In the Ubuntu terminal, as *root*, run:

```
wget https://raw.githubusercontent.com/fzampirolli/mctest/master/_setup-all.sh
```

Then replace `yourLogin` with the local username and run:

```
sed -i 's/\/home\/fz\//\/home\/yourLogin\//g' _setup-all.sh
source _setup-all.sh
pip install mysqlclient
```

After a few minutes, MCTest will be configured. To run it:

```
source /home/yourLogin/PycharmProjects/runDjango.sh
```

The system is then accessible at `http://127.0.0.1:8000`.

A.2 Creating an exam in MCTest

In MCTest, every exam is configured on an **Exam** screen, which brings together Classes, Topics (associated with a subject, such as PDI-VC), and Questions — each question belongs to a single topic. Besides the usual database attributes, the exam has its own parameters, such as the number of questions drawn and the number of variations generated.

For the two exams described in this appendix, a set of parameterized questions was defined per exam, from which a subset is randomly drawn for each variation, totaling **110 distinct variations** — one per student enrolled in the classes.

The **Create-Variations** button generates a new set of variations each time it is triggered. When questions are linked to VPL activities, the instructor receives by e-mail a `*linker.json` file containing all the test cases for the generated variations. The **Create-PDF** button generates, for each class, a PDF with the exams drawn per student, plus a `*students_variations.csv` file with each student's name, e-mail, and drawn variation.

> **! Warning**
>
> Each click of the **Create-Variations** button generates a new set of exam variations. After printing the PDF, it is essential **not to change the exam's attributes**, at the risk of invalidating the automatic grading of exams already printed or applied.

A.3 Creating a parametric question

A parametric question in MCTest combines three elements:

1. **Description** — the question statement, written in LaTeX, containing variables between `[[code:variable]]`, which are replaced by the value drawn for each student;
2. **Definition block** (`[[def: ... ]]`) — a Python code snippet, embedded in the question itself, responsible for drawing the parameters, computing the reference solution, and building the test cases;
3. **Test cases for Moodle** (`moodle_cases` block) — a JSON-serialized dictionary containing the expected inputs and outputs, consumed by the VPL activity.

A simplified example of a question definition (adapted from generating an `h × w` chessboard) is:

```
[[code:texto]]

\vspace{2mm}\noindent\textbf{Input Example:}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_inp]]
\end{verbatim}
\vspace{-2mm}\noindent\textbf{Output Example:}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_out]]
\end{verbatim}

\begin{comment}
[[code:moodle_cases]]
\end{comment}

[[def:
import json
import numpy as np
from topic.morph import mm # MUST include "topic." in MCTest

def chess(h, w):
    m = np.zeros((h, w), dtype='int')
    for i in range(h):
        for j in range(w):
            if (i + j) % 2:
                m[i][j] = 1
    return m

# PARAMETERS USED IN THE QUESTION DESCRIPTION
height = int(np.random.randint(5, 15))

# FULL STATEMENT, WITH THE height VALUE ALREADY INTERPOLATED
texto = (
    r"\vspace{2mm}\noindent\textbf{Description:}\vspace{-2mm} "
    r"Write a program that reads an integer \texttt{W}, representing the "
    r"width (number of columns) of a board, and prints that board in "
    r"the form of a chessboard, using the digits \texttt{0} and \texttt{1}. "
    r"The printed board will always have a height (number of rows) equal to "
    f"\\textbf{{{height}}} "
    r"and a width equal to the value \texttt{W} read from the input. "
```

```
    r"Considering position $(i, j)$ of the board (indexed from 0, with "
    r"$i$ representing the row and $j$ the column), the value printed at "
    r"that position must be \texttt{1} if $i + j$ is odd, and \texttt{0} otherwise. "
    r"Each row of the board must be printed on a separate line, with the "
    r"values of each column separated by a space."
)

inp_list, out_list, test_cases = [], [], 4
for i in range(test_cases):
    width = int(np.random.randint(500, 1500) / 100)
    inp = str(width) + '\n'
    out = mm.drawImg(chess(height, width)) + '\n'
    inp_list.append(inp)
    out_list.append(out)

cases = {}
cases['skills']      = ["matrices", "loops", "output formatting"]
cases['description'] = [{"text": latex_to_text(texto)}]
cases['input']       = inp_list
cases['output']      = out_list

moodle_cases = json.dumps(cases)

caso0_inp = cases['input'][0]
caso0_out = cases['output'][0]
]]
```

The `height` value is drawn once per variation (to appear in the statement), while `width` is drawn for each test case, increasing the robustness of automatic grading.

It is worth highlighting the dual role of the `texto` variable in this mechanism. It is at the same time:

- **the content of the printed statement**, inserted into the question's LaTeX description through the `[[code:texto]]` tag, appearing in the PDF generated by the **Create-PDF** button; and
- **an entry in the dictionary exported to Moodle**, through the `cases['description']` key, which becomes part of the `moodle_cases` JSON. It is precisely this field that the VPL activity shows the student when they open the question in Moodle to review it or submit a solution.

There is, however, an important difference between these two uses: the PDF is composed in LaTeX and therefore normally interprets commands such as `\textbf{}`, `\texttt{}`, or `$...$`. Moodle's VPL, on the other hand, does **not** process LaTeX — it expects plain text. That is why, when building `cases['description']`, `texto` is not inserted directly, but passed through MCTest's own utility function `latex_to_text()`, which removes (or converts) the LaTeX commands and symbols from the statement, producing an equivalent plain-text version. Thus, `[[code:texto]]` in the PDF still receives the original `texto`, with all its LaTeX formatting, while `cases['description']` receives `latex_to_text(texto)`, ensuring that the statement shown in Moodle is readable even without LaTeX support.

Since `texto` is built inside the `[[def: ...]]` block itself — in the same scope where `height`, `width`, and the other parameters are drawn —, it is recalculated for every variation. This guarantees that the statement seen by the student in Moodle is **always identical** to the one printed on the paper exam, even when the text changes from student to student along with the drawn values. Section A.4 revisits this point with a real case, in which the statement is significantly longer and describes, besides the numeric parameters, the visual scenario itself generated for each variation.

Figure A.1 shows the chessboard question actually generated by MCTest for one of the variations, with the statement already containing the drawn value of `height` and the input/output example corresponding to the first test case:

Clicking **Create-PDF** on the question screen generates a new variation with each click, allowing the statement to be checked before publishing.

1. **Descrição:** Escreva um programa que leia um número inteiro `W`, representando a largura (número de colunas) de um tabuleiro, e imprima esse tabuleiro no formato de um tabuleiro de xadrez, usando os dígitos `0` e `1`. O tabuleiro impresso terá sempre altura (número de linhas) igual a **6** e largura igual ao valor `W` lido da entrada. Considerando a posição (i, j) do tabuleiro (indexada a partir de 0, com i representando a linha e j a coluna), o valor impresso nessa posição deve ser `1` se $i + j$ for ímpar, e `0` caso contrário. Cada linha do tabuleiro deve ser impressa em uma linha separada, com os valores de cada coluna separados por um espaço.

Exemplo de Entrada:

```
9
```

Exemplo de Saída:

```
0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1
```

Figure A.1: Chessboard question generated by MCTest, illustrating the parameterized statement with the `height` value drawn for the variation and the input/output example of the first test case.

For the two exams of the course, this mechanism was used more broadly: each parametric question defines not only numeric values but also, in some cases, small structural variations in the statement (for example, which cardinal direction — North, South, East, West — should be evaluated), making it harder to find ready-made solutions online or via generative AI tools.

A.4 Real example: a question from Mock Exam 4

To illustrate the mechanism described in the previous section with a concrete case, the following is a question actually applied in **Mock Exam 4** (topic *im4-Morphological Operators*, difficulty 1, Bloom's taxonomy "remember"), a parameterized programming question with Moodle+VPL integration.

The question asks the student to write a program capable of:

- reading a binary image, corrupted by salt-and-pepper noise, containing at least two isolated geometric objects;
- applying morphological filtering (opening followed by closing) to remove the noise;
- extracting, from the filtered image, the geometric measurements of each object (area, perimeter, center, bounding box, circularity, solidity, and number of vertices), using the helper function `mm.measure`;
- sorting the objects by position (bounding-box X coordinate, with Y as tiebreaker) and reassigning the identifiers sequentially;
- printing the cleaned matrix and the measurement table in the format expected by the automatic grader.

In the corresponding `[[def: ... ]]` block, a scene generator (`gerarCena`) randomly draws the height and width of the image, the type, size, and position of each object (square, rectangle, triangle, or block), ensuring they do not overlap, and then adds salt-and-pepper noise with a 3% probability per pixel. Ten distinct test cases are generated this way for each exam variation, and the input/output pair of the first case is reused in the statement itself as an example for the student. The morphology library (`mm`) is imported directly from `morph.py`, also available as MCTest's own utility module.

In summary, the statement asks the student for a program that:

1. reads, on the first line, two integers H and W (image height and width);
2. reads the following H lines of the binary matrix (0s and 1s separated by spaces), using `mm.readImg(H, W)`;
3. applies morphological filtering to remove the salt-and-pepper noise;
4. prints the already filtered matrix, as 0s and 1s separated by spaces;
5. extracts the geometric measurements of the objects with `mm.measure(imgLimpa)`;
6. sorts the objects and prints the measurement table in the expected format.

The statement also brings two important observations for automatic grading: (i) the area computed by

OpenCV (`cv2.contourArea`) corresponds to the continuous polygon delimited by the centers of the border pixels, and is therefore smaller than the simple count of pixels equal to 1 (`np.sum`); and (ii) the list of objects must be sorted in ascending order by the bounding-box X coordinate (`bbox[0]`), using the Y coordinate (`bbox[1]`) as tiebreaker, with the IDs reassigned sequentially from 1 to N after sorting.

As in the example in Section A.3, the statement text here is also not fixed: it is built inside the `[[def: ... ]]` block itself, in the Python variable `texto`, and plays the same dual role already described — it feeds the PDF via `[[code:texto]]` and is exported, already converted by `latex_to_text()`, into `cases['description']` inside `moodle_cases`, which is the version shown to the student in Moodle's VPL. The difference is that, in this real case, it is the **visual scenario** (noisy image, number and position of objects) that changes from student to student with each variation, while the wording of `texto` stays fixed across variations, since only the numeric data (image and test cases) are drawn. Ideally, the wording would also vary with each generation, as in the previous example, where the `height` value was interpolated directly into the description text. The actual structure used (with scenario-drawing snippets omitted for brevity) is:

```
[[code:texto]]

\vspace{2mm}\noindent\textbf{Input Example:}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_inp]]
\end{verbatim}
\vspace{-2mm}\noindent\textbf{Output Example:}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_out]]
\end{verbatim}

\begin{comment}
[[code:moodle_cases]]
\end{comment}

[[def:
import json
import numpy as n
import cv2
from topic.morph import mm # MUST include "topic." in MCTest

# ... scene, noise, and 10 test-case generation (inplist/outlist) ...

# FULL STATEMENT, WITH EXPLANATION OF AREA AND SORTING
texto = (
    "A binary image corrupted by salt-and-pepper noise contains at least two isolated
    geometric objects.\n\n"
    "Write a program that:\n"
    "1. Reads two integers ..."
)

cases = {}
cases['skills']      = ["mathematical morphology", "noise removal", "mm.measure",
"tabulation"]
cases['description'] = [{"text": latex_to_text(texto)}]
cases['input']       = np.array(inplist).tolist()
cases['output']      = np.array(outlist).tolist()

moodle_cases = json.dumps(cases)
caso0_inp = cases['input'][0]
caso0_out = cases['output'][0]
]]
```

Below is the photograph of the question actually generated and printed for one of the 110 variations of Mock Exam 4, showing the statement, the noisy input image, and the expected output example (filtered image and

measurement table):

1. Uma imagem binária corrompida por ruído sal e pimenta contém no mínimo dois objetos geométricos isolados.

Escreva um programa que: 1. Leia dois inteiros na primeira linha representando a altura (H) e largura (W) da imagem. 2. Leia as H linhas seguintes da matriz binária (valores 0 e 1 separados por espaço) usando mm.readImg(H, W). 3. Aplique a filtragem morfológica para eliminar o ruído sal e pimenta. 4. Imprima a matriz limpa diretamente em valores 0 e 1 (0s e 1s separados por espaço). 5. Extraia as medições geométricas dos objetos usando mm.measure(imgLimpa). 6. Ordene a lista de objetos e imprima a tabela formatada das medidas.

Observação sobre a área: A área calculada pelo OpenCV (cv2.contourArea) mede o polígono contínuo delimitado pelos centros dos pixels de borda, resultando em um valor menor do que a simples contagem discreta de pixels 1 (np.sum).

Regra de Ordenação: A lista de objetos deve ser ordenada em ordem crescente pela posição X do retângulo delimitador (bbox[0]), utilizando a posição Y (bbox[1]) como critério de desempate caso dois objetos comecem na mesma coluna (medidas.sort(key=lambda m: (m['bbox'][0], m['bbox'][1]))). Em seguida, reatribua os IDs sequencialmente de 1 a N.

Exemplo de Entrada:

```
12 43
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 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 0 0 0 0 0 1 0 0 0 0 1 1 1 1 1 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 0 0 0 0 0 0 0 0 0 0 0 0 1 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 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 0 0 0 0 1 1 0 0 0 0 0 0 0 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 1 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 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 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Exemplo de Saída:

```
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 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 0 0 0 0 0 0 0 0 0 1 1 1 1 1 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 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
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 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 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 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 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 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 0 0 0 0 0 0
 id area perimeter    cx    cy   x   y   w   h circularity solidity vertices
  1   20      18.0   6.5   2.0   4   0   6   5        0.78    1.000        4
  2   20      18.0  26.0   2.5  24   0   5   6        0.78    1.000        4
```

Figure A.2: Real question generated by MCTest for Mock Exam 4 — "Morphological Operators" topic.

A.5 Final remarks

This appendix summarized the steps to install MCTest, create exams, and build parametric questions with Moodle VPL integration. The combined use of variables drawn in the description (`[[code:...]]`) and Python code embedded in the question itself (`[[def: ... ]]`) makes it possible to generate hundreds of exam variations from a single template, each graded automatically and consistently — with the `texto` variable, in particular, guaranteeing that the same statement is shown both in the printed PDF and when the question is reviewed by the student in Moodle. The real example of Mock Exam 4 (Section A.4) illustrates how this mechanism is used in practice to generate programming questions with visual scenarios that differ per student. Appendix B details how these variations are published and graded in Moodle, and Appendix C describes how to restrict access to exams using SEB.

Appendix B

About Moodle (VPL)

This appendix describes the configuration of a **VPL (*Virtual Programming Lab*)** activity in Moodle, the *plugin* used for submitting and automatically grading the Programming Exercises (EPs), the biweekly mock exams, and the exams described in this book. The Moodle version used was **4.5**.

B.1 Configuring a VPL activity

Creating an exam in MCTest with VPL integration (see Appendix A) generates two files that must be renamed to `linker.json` and `students_variations.csv`. These files, together with `morph.py`, must be added to the VPL activity's **Execution Files**, along with all the files contained in `VPL_modification/V14-AI-feedback.zip`, available in the MCTest repository at https://github.com/fzampirolli/mctest. All of them must be marked with the **"*Files to keep when running*"** option.

In short, the VPL activity then knows, for each student, which variation of the question was drawn (via `students_variations.csv`) and which set of test cases corresponds to it (via `linker.json`). This allows grading to be individualized, even when dozens of students solve the same activity with different variations.

B.2 Publishing EPs as VPL activities

The Programming Exercises (EPs) presented at the end of each chapter of this book can also be published as VPL activities, following the same mechanism. The typical classroom usage flow was:

1. Opening the chapter's two Colab *notebooks* (theoretical and practical), with emphasis on the interactive simulators;
2. Running the code blocks, changing parameters to reinforce understanding of the concepts;
3. In lab classes, submitting the EP answers directly in the corresponding VPL activity, with immediate feedback on passing or failing the test cases.

If the VPL activity is not generated by MCTest, the `linker.json` and `students_variations.csv` files are not needed. However, to use AI *feedback* (see Appendix D), the files contained in `VPL_modification/V14-EP0_VPL-TEMPLATE.zip`, available in the same repository (https://github.com/fzampirolli/mctest), must be added, following the procedure described in Section B.1.

> **Reusing test cases**
>
> The `.cases` files used in local validation (the `TestSuite` class, described in the body of this book) are compatible with the format expected by VPL, allowing the same set of tests to be reused both in EP development and in automated grading in Moodle.

In the biweekly mock exams — applied with SEB (see Appendix C) and worth a 5% bonus on the final grade —, VPL activities were used with EPs similar to those in the regular assignment lists, but graded under internet-access restrictions, reinforcing the assessment nature of the activity.

B.3 Final remarks

This appendix described the configuration of VPL activities in Moodle for automatically grading EPs, mock exams, and parameterized exams. The combination of MCTest (generating variations and test cases), VPL (submission and grading), and SEB (access restriction) forms the tripod on which the automated, individualized assessment used throughout the course rests.

Appendix C

Using SEB in Biweekly Mock Exams and Assessments

Besides the two exams with parameterized questions (see Appendix A), the ***Safe Exam Browser*** **(SEB)** was also used to restrict access during the **biweekly mock exams** — VPL (*Virtual Programming Lab*) activities with Programming Exercises (EPs) similar to those in the regular assignment lists, worth a bonus of up to 5% on the final grade —, as well as the other practical assessments.

SEB ensures a secure assessment environment by blocking access to unauthorized external resources on the lab machines. For the implementation adopted in this work, **Windows 10** and SEB version **3.7.1.704**, available at https://safeexambrowser.org/, were used, and must be pre-installed on every lab workstation.

The procedure for configuring the SEB environment file and linking it to the VPL activity in Moodle is described below.

C.1 Configuring the application in *SEB Tool*

To generate the `.seb` configuration file that will be distributed to students, open the *SEB Tool* utility and follow the steps below.

1. ***General***
   - In the ***Start URL*** field, enter the direct URL of the VPL activity in Moodle.
   - *Note:* if it is necessary to navigate to more than one activity within the same course, enter the course's main URL and, on the ***Browser*** tab, check the ***Allow navigation back/forward in exam*** option.
2. ***Network***
   - Include the URLs allowed for navigation. To restrict access to the desired activity, use a filter expression in the pattern `/mod/vpl/*?id=4624*`, where the final number corresponds to the VPL activity's identifier in Moodle. In Table C.1, the last row contains the URL of the course's main page (identifier `475`), but it can also be replaced by the URL of a section containing several activities. This setting is optional and useful when there is more than one VPL activity. In that case, the first row of the table should be repeated for each assessment activity that should remain accessible during the *exam*.
   - **General settings on the *Filter* tab**:
     ☒ *Activate URL filtering*
     ☐ *Filter also embedded content*

Table C.1: URL filter expressions used in the SEB configuration.

Active	*Regex*	*Expression*	*Action*
[x]	[ ]	`https://moodle.ufabc.edu.br/mod/vpl/*?id=4624*`	*Allow*
[x]	[ ]	`https://moodle.ufabc.edu.br/login/index.php*`	*Allow*
[x]	[ ]	`https://moodle.ufabc.edu.br/course/switchrole.php*`	*Allow*
[x]	[ ]	`https://mctest.ufabc.edu.br/compare`	*Allow*

Active	*Regex*	*Expression*	*Action*
[x]	[ ]	`https://moodle.ufabc.edu.br/enrol/index.php?id=475`	*Allow*

3. ***Security***
   - If the lab uses a projector or a secondary monitor, set the ***Maximum allowed number of connected displays*** field to a value greater than 1, to prevent SEB from blocking the display.
4. ***Config File***
   - Click ***Save Settings As...*** and save the file with the `.seb` extension.
5. ***Exam***
   - Check the ***Use Browser Exam Key and Configuration Key*** option.
   - Copy the code generated in the ***Browser Exam Key*** field. *Important:* copy this key only **after** saving the file in the previous step.

C.2 Linking the SEB key to the VPL activity in Moodle

After generating the file and obtaining the *Browser Exam Key*, access Moodle to apply the restrictions.

1. On the VPL activity page, click the gear icon (**Settings**).
2. Go to ***Configuration*** → ***Submission restrictions*** and click ***Show more***.
3. Change the ***SEB browser required*** option to ***Yes***.
4. Paste the copied key into the ***SEB exam key(s)*** field.

C.3 Restriction by IP address (optional)

To ensure that submissions come exclusively from the lab computers, SEB can be combined with Moodle's network filter, by filling in the ***Allowed submission from net*** field.

- **Consecutive ranges:** use a hyphen to define the range. Example: `111.11.11.1-62` (allows IPs from `111.11.11.1` to `111.11.11.62`).
- **Non-consecutive IPs:** list the individual addresses separated by commas.

C.4 Final remarks

The integration of SEB with VPL in Moodle creates a simple flow for the student: just double-click the `.seb` file provided by the instructor for the secure browser to open directly on the configured activity.

The combination of the *Browser Exam Key* with IP-range restriction on the lab machines provides a robust solution for *online* programming assessments, hindering improper access and contributing to the integrity of the assessment process.

Appendix D

Generating Course Material with *Gemini Notebook*

This appendix describes the use of ***Gemini Notebook* (*NotebookLM*)** as a tool supporting the preparation of course material, complementing the use of AI for reviewing text and code, mentioned in this book's preface.

D.1 Conceptual videos and EP-solving videos

For each chapter, a project was created in *Gemini Notebook* using the full PDF of the book and the `morph.py` file as sources. From these sources, *Gemini Notebook* automatically generated a short video (7 minutes on average), used at the start of classes. Chapters alternated between two types of video:

- **Conceptual video**, presenting the chapter's theoretical foundations, generally shown in the first class dedicated to the topic;
- **EP-solving video**, presenting a solution strategy for the Programming Exercises, shown in the following class, when most EPs were solved together with the class.

> **Role of the video in class**
>
> The video generated by *Gemini Notebook* does not replace the instructor's explanation. It works as a brief introduction and motivation for the topic, giving the student context before the *slides* are presented and the Colab *notebooks* are opened.

D.2 Supporting *slides*

Complementing the videos, *Gemini Notebook* also generated, from the same sources (the book's PDF and the `morph.py` file), a set of approximately 12 *slides* per chapter, covering both the theoretical concepts and the practical part. Generation used a chapter-specific *prompt*, restricting the scope of content to be synthesized and avoiding both anticipating topics from later chapters and reproducing long passages of the book without didactic adaptation.

After the *slides* were presented, class continued with the opening of the chapter's two Colab *notebooks* (theoretical and practical), emphasizing the interactive simulators and running the code blocks with parameter changes.

D.3 Main material and final remarks

Generating videos and *slides* with *Gemini Notebook*, based on the book itself and the `morph.py` library, made it possible to standardize the opening of classes and reduce the time spent preparing course material, without dispensing with the instructor's curation in crafting the *prompts* and conducting classroom activities.

The videos and *slides* produced by *Gemini Notebook* are **motivational** in nature and serve as a starting point for discussing the concepts presented in each chapter. Like any AI-generated content, they may contain

occasional inaccuracies or errors. In some cases, these inaccuracies are intentionally exploited during class to check whether students are critically following the presentation and can identify conceptual inconsistencies.

The course's official content, however, is the material produced by the method described in this book's preface, made available in three complementary formats: **HTML**, for browsing with interactive simulators; **PDF**, for reading and printing; and **Jupyter** *notebooks* (`.ipynb`), for running and experimenting with the code. All material generated by *Gemini Notebook* should be understood as a complement to this main content.

As with the other uses of generative AI described in this book, responsibility for the technical quality, conceptual correctness, and pedagogical adequacy of the material remains with the instructor.

Appendix E

Using AI in Student Assessment: the vpl-ai-feedback Project

This appendix describes **vpl-ai-feedback**, an open-source system developed to support the grading of code submissions sent to VPL (Moodle) through *feedback* generated by Artificial Intelligence (AI). Unlike a traditional automatic grader — which only compares outputs against test cases —, **vpl-ai-feedback** sends the student's code, together with the question rubric, to a language model (LLM), which returns a qualitative analysis per criterion, in the style of ***Socratic* feedback**: the student receives comments pointing out what was done correctly, what can be improved and why, in both **formative** activities (practice exercises) and **assessment** activities (mock exams) in VPL.

This Socratic *feedback* mechanism for VPL activities was proposed by Zampirolli et al. (2026). The study describes the integration of AI into the assessment process for parameterized programming exercises in VPL/Moodle, using a set of seven LLMs adopted to analyze student code and generate automated *feedback* on every grading request, with preliminary results indicating a positive student perception regarding idea generation, clarity of explanations, and learning autonomy.

The rest of this appendix details the practical implementation of these ideas in the **vpl-ai-feedback** project — whose code is available at https://github.com/fzampirolli/vpl-ai-feedback — and specifically describes its use in the three PDI-VC classes between May and August 2026.

> ⚠ Warning: AI does not replace instructor judgment
>
> Using AI to **grade** students **is not recommended as the sole source of a grade**, since language models can **hallucinate** — that is, produce plausible but incorrect assessments, criteria, or justifications, even for correct or incorrect code. An LLM may give a high grade to code with a subtle bug, or a low grade to correct code written in an unusual way, simply because the generated explanation "sounds" coherent. **AI should be used only as a complement to assessment**, never as a substitute for the instructor. Appendix A and Appendix B describe the objective grading mechanisms (test cases) that remain the basis of the official grade; this appendix deals exclusively with the use of AI as an additional layer of qualitative *feedback*.

E.1 Context of use

vpl-ai-feedback was used in three sections of the **PDI-VC** course, taught between May and August 2026, totaling approximately one hundred students. AI *feedback* was used in three complementary ways:

1. **Continuous *feedback* in formative activities** — practice exercises submitted to VPL throughout the course, in which the student could resubmit code as many times as desired. On each submission, they received qualitative AI-generated *feedback*, in addition to the objective grading performed by VPL, described in Zampirolli et al. (2026).
2. **Complementary *feedback* in mock exams (bonus assessment activities)** — four mock exams held throughout the term, planned in the syllabus as bonus activities, each worth up to 5% of the final grade. Applied approximately 30 minutes before the end of the biweekly lab practical classes, the mock

exams used **vpl-ai-feedback** to generate an individual assessment rubric, sent by e-mail together with a comparative summary between the grade assigned by Moodle/VPL and the assessment produced by the AI.

3. **Complementary *feedback* on Exams 1 and 2 (official assessments)** — the term's two formal exams, each worth a larger share of the course's final grade, also began using **vpl-ai-feedback** after they were applied. Unlike the mock exams (bonus), here the official grade is already determined by VPL's objective grading and/or the instructor's manual review before results are released; the AI-generated rubric is sent to the student only afterward, as supporting material for reviewing their own performance — reinforcing, also in the higher-weight assessments, the separation between *official grade* and *complementary feedback* detailed in Section D.7.

An opinion survey, answered by 102 students before the system was adopted, indicated high acceptance of the proposal. Only 2 students gave a rating of 1 (rejection) and 7 gave a rating of 2 regarding interest in receiving automatic AI code *feedback*. Another 19 declared themselves indifferent (rating 3), while the majority — 38 and 37 students — gave ratings of 4 and 5 (high approval), respectively, totaling the 102 respondents. This result motivated the adoption of the system as a complement to the formative process, not as a substitute for human grading.

! The official grade was never the AI's grade

It is essential to stress that, as defined in the syllabus, **the grade used both for calculating each mock exam's bonus and for the official grade of Exams 1 and 2 was always the grade assigned by VPL** (based on the test cases) and/or by the instructor's manual review — and **not** the grade suggested by the AI. The rubric generated by **vpl-ai-feedback** was sent to the student only as supporting learning material — never as a criterion for assigning a grade, whether in bonus activities or official assessments. This separation between *official grade* (VPL/instructor) and *complementary feedback* (AI) is the central principle of this appendix and was revisited in Section D.8.

E.2 Project architecture

vpl-ai-feedback is an asynchronous Python *script* organized as follows:

```
.
  config.yaml              ← configuration (credentials, provider, weights) - NEVER versioned
  config.yaml.example      ← public configuration template
  main.py                  ← main script (async), orchestrates the whole process
  run.sh                   ← execution wrapper (bash run.sh config.yaml)
  gerar_relatorio.py       ← converts the consolidated *_ALL.txt into CSV
  enviar_email.py          ← sends the rubrics by e-mail to each student
  providers/                ← LLM API clients
      __init__.py          ← client factory (Factory)
      base.py               ← retry, backoff, and fallback logic between models
      groq.py
      deepseek.py
      gemini.py
  core/                      ← business logic
      grader.py              ← identifies files by question, builds the prompt, calls the LLM
      utils.py
  <class_folder>/          ← submissions downloaded from Moodle VPL
      Student name - login/
          <timestamp>/                 ← student's latest submission
              Q1.py                    ← pattern for exams with multiple questions
              Q2.py
              rubrica.txt              ← generated by the LLM (only if the assessment is
   valid)
          <timestamp>.ceg/
```

```
            execution.txt            ← objective grade assigned by VPL
    ...
```

The system accepts three LLM providers — **Groq**, **DeepSeek**, and **Gemini** — configurable in `config.yaml` via the `llm.provider` key. Each provider can have **multiple registered models**; on each call, the list is **shuffled**, distributing the load among students processed in parallel and reducing the chance of rate-limit errors (HTTP 429). When a model fails, the system retries up to three times before moving to the next one in the list, respecting the wait time suggested by the provider. Authentication errors (401) or insufficient-balance errors (402) immediately abort the attempt with that provider.

An important aspect of the project concerns data privacy. **The student's name, e-mail, login, student ID, and national ID are never sent to the LLM API**. Only the following are transmitted for processing: the name of the file containing the source code; the code itself (with comments removed); the rubric *prompt* — which contains no personal data —; and, when available in the `execution.txt` file generated by VPL, the official statement of the question drawn for that student and the raw result of the test cases actually executed (input, output produced by the code, and expected output). These last two items, although extracted from a file specific to each student, also contain no personal identification — only the question text and the input/output values of the automated tests —, and are used as objective evidence to reduce the risk of the AI incorrectly identifying the question type or assessing the code's logic based solely on a static reading (Section D.3).

Even so, it is important to recognize that the three currently supported providers — Groq, DeepSeek, and Gemini — are commercial services operated by foreign companies, whose models run on infrastructure outside the country and are subject to usage, availability, and cost policies beyond the institution's control. This dependence on external providers carries risks that go beyond one-off code grading: price changes, model discontinuation, temporary service unavailability, unilateral changes to terms of use, and, in more sensitive cases, geopolitical access restrictions could compromise the continuity of a grading system that becomes structurally dependent on these APIs. For this reason, it is strategic for Higher Education Institutions (HEIs) such as UFABC to invest in developing and hosting their **own AI providers** — for example, open-weight models run on local infrastructure or a sovereign cloud —, reducing dependence on external services, especially from other countries, and expanding institutional control over cost, availability, student data privacy, and long-term pedagogical continuity. The design of **vpl-ai-feedback** already favors this path: the `providers/` architecture was built as an extensible factory, in which a new provider — including a model hosted locally by the institution itself, with a compatible interface — can be added with low integration effort.

E.3 The universal *prompt* and the two-stage rubric

Each exam type is described in a *prompt* file (for example, `Simulado4.txt`), referenced in `grading.prompt_file` in `config.yaml`. This file instructs the LLM to perform the assessment in **two stages**:

1. **Question identification** — the LLM determines the specific operation type drawn for that student (useful when there are parameterized questions drawn and shuffled per student, as described in Appendix A);
2. **Applying the corresponding rubric** — the LLM assesses the code against scored criteria, each with a maximum point range, ultimately returning a grade in the format `FINAL GRADE: X/WEIGHT`.

For parameterized exams, the *prompt* file usually contains **several parallel rubrics**, one for each possible question type, delimited by markers such as `[START_RUBRICA_TIPO_A]` / `[END_RUBRICA_TIPO_A]`. The LLM first identifies which type was drawn for that student and applies **only** the corresponding rubric, ignoring the others — which prevents criteria from a different type from contaminating the grade.

When the exam has more than one question per VPL activity, the code files must follow the pattern `Q1.*`, `Q2.*`, etc. — one file per question, with the extension free according to the language chosen by the student —, and each question has an individually configurable weight in `grading.weights` in `config.yaml`. When the exam has a single question and was not generated by MCTest, the system accepts any file name with a supported extension, as long as it is the only code file present in the submission folder.

In the case of Mock Exam 4 (topic *Morphological Operators*), for example, one of the possible types defined three criteria, with a total weight of 100 points for that question:

Criterion	Description	Maximum score
1 — Data input	Reading `H`, `W`, and the binary matrix (`mm.readImg`)	25 pts
2 — Morphological processing	Opening + Closing (`mm.sebox()`), `mm.measure`, sorting by `bbox[0]`/`bbox[1]`, and reassigning IDs	50 pts
3 — Output and formatting	Displaying the cleaned image (`mm.drawImg`) and the formatted measurement table	25 pts

The *prompt* also requires a **mandatory response format** (line width, section order, the `FINAL GRADE:` marker), which makes automatic grade extraction and the consolidated report generation more reliable — although, as discussed in Section D.8, this does not eliminate the risk of semantic error in assessing the content.

E.4 Two additional layers of evidence

To reduce the risk of the LLM hallucinating the question type or the correctness of the code — the central risk discussed in Section D.7 —, **vpl-ai-feedback** started providing the AI with two sources of objective evidence, automatically extracted from VPL's own `execution.txt`, in addition to the source code:

- **Official question statement.** Since questions are usually drawn and shuffled per student in MCTest (Appendix A), each student's `execution.txt` already contains, in the "Question summary" field, the exact text of the question they were assigned. The system automatically extracts this excerpt and sends it to the LLM as **primary** evidence for identifying the question's type/variation — more reliable than inferring solely from a static reading of the code, especially in incomplete submissions or those ambiguous between two close types (for example, "flat" erosion *versus* "weighted" erosion). When the submitted code diverges from what the statement asks, the system does not ignore the statement nor silently "correct" the reading: it keeps the identification based on the statement, lowers the assessment's confidence to "Low," and makes the divergence explicit in the *feedback* sent to the student.
- **Actual VPL execution result.** When available, the system also sends the LLM the test cases actually run on that code — input, output produced, and expected output — as objective evidence of correctness, to be used to confirm or refute the reading of the logic before scoring the processing and output criteria, instead of the AI relying solely on mentally simulating the code (which is especially error-prone in loops with `min`/`max`, index offsets, or Otsu threshold computation, where subtle errors go unnoticed in a static reading).

In addition, the LLM is instructed to explicitly state its **confidence level** (High, Medium, or Low) in identifying each question type, with justification when confidence is not High. This value is automatically extracted and feeds a `Revisar_Manualmente` (Review Manually) column in the consolidated CSV report (Section D.4), allowing the instructor to prioritize exactly the cases where the AI itself recognizes greater uncertainty — instead of treating all of a class's assessments as equally reliable.

E.5 Execution flow

The typical run, done via `bash run.sh config.yaml`, follows these steps:

1. Loads `config.yaml` and locates the submissions folder (`paths.student_base_dir`);
2. For each student, identifies the **latest submission** (the folder with the most recent *timestamp*);

3. Extracts the objective Moodle grade from `*.ceg/execution.txt`;
4. For each question of the exam, locates the code file (`Q1.*`, `Q2.*`, or the single file, for an exam not generated by MCTest), builds the *prompt* (code + rubric), and sends it to a model drawn from the configured list;
5. Processes all students **in parallel**, with concurrency control;
6. Generates, per student, the `rubrica.txt` file — **only if the AI returns at least one valid assessment** for the questions that have code; otherwise, the file is not saved, forcing a retry on the next run;
7. Consolidates all `rubrica.txt` files into a single `*_ALL.txt` and generates a `*_relatorio.csv` report, comparing the Moodle grade, the AI grade, and the difference between them, per question and in total.

Situation	Is `rubrica.txt` saved?
All questions with code successfully assessed	✅ Yes
AI failed on at least one question with code	❌ No — reprocessed on the next run
Student submitted no code file	❌ No
Question type identified as UNKNOWN	❌ No

E.6 Illustrative example: rubric for a three-question exam

ℹ About the origin of this example

The following excerpt **does not reproduce any specific student's code**. It is an example reconstructed by the author, reproducing an error *pattern* repeatedly observed across PDI-VC classes — confusion between a 2D morphological function (`mm.ero`/`mm.ero0`) and a 1D operation on signals — without citing or identifying any actual submission. This choice avoids any risk of infringing a student's copyright over their own source code, even anonymized, since authorship of the work remains with the original author even when name and *login* are removed.

The following excerpt illustrates the output generated by **vpl-ai-feedback** for a student on a three-question exam (Q1, Q2, Q3), using the `deepseek-chat` model. The comparative summary appears at the top of the file, followed by the official statement of each question, the submitted code, and the assessment per criterion.

```
SUMMARY - AI (deepseek-chat)  x  MOODLE

Total weight : 100 pts

Q1 (AI) :  3.3 / 33 pts
Q2 (AI) :  6.7 / 33 pts
Q3 (AI) : 33.3 / 34 pts

Moodle : (Q1=33 + Q2=0 + Q3=34) = 67 pts
AI     : 43.3 / 100 pts

Difference (AI - Moodle): -23.7 pts

Q1: identified type = B  | confidence = Low (code does not
  match the question statement)  REVIEW
Q2: identified type = A  | confidence = High (official statement
  confirms the type and variation)
Q3: identified type = C1 | confidence = High (official statement
  confirms 2D erosion with integrated OpenCV)
```

Q1 — code does not match the statement (low confidence): the question's official statement,

extracted from `execution.txt`, asked for a **1D** morphological erosion on a signal (reading N, M, the signal, and structuring element B, followed by the minimum of the active neighbors). The submitted code, however, only reads two integers and calls `mm.ero0` — a **2D** erosion function for images, unrelated to the requested operation.

```
Identified type: TYPE B (1D Erosion - flat)
Confidence: Low - code does not match the question statement (uses
mm.ero0, a 2D erosion function, instead of implementing flat 1D
erosion on a signal)

Criterion 1 - [3.33/10.00 pts]:
- The code reads two integers on a single line (which would correspond
  to N and M), but does not read the signal or the structuring element
  B as 1D vectors. Instead, it treats both as images via mm.readImg,
  which does not match the expected reading of a 1D signal and of B on
  separate lines.

Criterion 2 - [0.00/16.67 pts]:
- The operation applied is mm.ero0(img, element), a 2D erosion for
  images, not the 1D signal erosion requested in the statement. The
  logic of the minimum of active neighbors with border cropping is not
  implemented.

Criterion 3 - [0.00/6.67 pts]:
- The output is a 2D image via mm.drawImg, not the resulting 1D signal
  on a single line with integer values separated by spaces, as
  required.

FINAL GRADE: 3.33/33.33

Comments for the student: The submitted code does not match what the
statement asks. The drawn question requests a 1D morphological erosion
on a one-dimensional signal; the code uses a 2D erosion function for
images. Re-read the assigned statement carefully before choosing the
implementation.
```

Q2 — syntax error prevents execution (high confidence, low grade due to a real failure): here the AI correctly identified the type (Otsu thresholding), supported by the official statement, but the code contained an indentation error that prevented execution — evidenced both by the static reading and by the absence of valid output in the VPL tests.

```
Identified type: TYPE A (Thresholding - manual T + Otsu)
Confidence: High - official statement confirms the type and variation

Criterion 1 - [6.67/10.00 pts]:
- Correctly reads the three parameters L, C, and T on separate lines
  and the input image. However, the definition of a helper thresholding
  function is incorrectly indented within the global scope, causing an
  indentation error that prevents the program from running completely.

Criterion 2 - [0.00/16.67 pts]:
- The fixed-threshold T binarization is correctly structured, but the
  Otsu threshold computation depends on the malformed helper function,
  which references a variable out of scope. The indentation error
  prevents any processing.

Criterion 3 - [0.00/6.67 pts]:
- Due to the indentation error, no output is produced.

FINAL GRADE: 6.67/33.33
```

```
Comments for the student: The overall structure is correct (reading L,
C, T, and the image), but there is an indentation error that prevents
execution. Review the Python syntax before submitting and test the code
locally before sending it to VPL.
```

Q3 — function mismatch, but functional equivalence confirmed by real evidence (high confidence, fair grade): the statement explicitly asked for the use of `mm.ero` (OpenCV-integrated version), but the code uses `mm.ero0` (manually implemented flat version). Since `execution.txt` showed all 10 VPL test cases passing, the AI did not penalize the function mismatch — it recognized the functional equivalence demonstrated by the real tests, but still noted the observation in the *feedback*.

```
Identified type: TYPE C1 (2D Erosion - flat, no weights)
Confidence: High - official statement confirms 2D erosion with
integrated OpenCV (mm.ero), and the code implements flat erosion via
mm.ero0

Criterion 1 - [10.00/10.00 pts]:
- Correctly reads height, width, the image matrix, and the dimensions
  and values of structuring element B.

Criterion 2 - [16.67/16.67 pts]:
- The statement asked for the use of mm.ero (integrated OpenCV), but
  the code uses mm.ero0 (flat, no weights). Despite the function
  mismatch, the operation was correctly implemented with the structuring
  element read: all 10 tests run on VPL passed completely (10/10),
  confirming functional equivalence for the tested cases.

Criterion 3 - [6.67/6.67 pts]:
- Prints the resulting image in the correct format; the tests confirmed
  the correct output.

FINAL GRADE: 33.34/33.33

Comments for the student: The code is correct and functional, with all
10 tests passing. Note: the statement explicitly asked for mm.ero
(OpenCV version), but you used mm.ero0 (flat version). Although the
results were equivalent in the tests, follow exactly the function
requested in the statement in future exams.
```

Note that, in this composite case, the AI assigned **23.7 fewer points** than VPL's objective grade, driven mainly by Q1. The aggregate difference alone would already be a signal for investigation (Section D.7), but the system itself hands the instructor the likely cause of each question: the `Revisar_Manualmente` column of the CSV is marked "YES" whenever at least one of the student's questions receives low confidence, with the specific reason recorded in the corresponding confidence column — reducing the time the instructor needs to spend finding, among a hundred students, the few cases that actually deserve manual attention before the official grade is assigned. The case of Q3, in turn, illustrates the opposite care: the divergence between statement and code did **not** result in undue penalization, because the real execution evidence confirmed the functional equivalence — exactly the behavior that Section D.3 describes for these two layers of evidence.

E.7 Sending *feedback* by e-mail

After the rubrics are generated, the `enviar_email.py` *script* sends each student an e-mail with `rubrica.txt` attached. The e-mail body is defined in `config.yaml`, under `templates.corpo`, and is interpolated with `{nome_pasta}` and `{login}`:

```yaml
email:
  smtp_server: smtp.ufabc.edu.br
  smtp_port: 587
  from_address: professor@ufabc.edu.br
  password: "YOUR_PASSWORD_HERE"     # never version real credentials
  use_tls: true

templates:
  assunto: "LLM-Generated Rubrics and Grading - Mock Exam - {login}@aluno.ufabc.edu.br"
  corpo: |
    Dear {nome_pasta},
    ...
```

! Credential security

`config.yaml` contains real secrets (SMTP password, API keys). It must be listed in the project's `.gitignore` and **never** be published, shared, or versioned — including in e-mail attachments, screenshots, or public repositories.

A real example of an e-mail sent to a student (data **anonymized**, with name and login replaced by a fictitious case) is reproduced below, to illustrate the didactic tone and the caveats made explicit to the student:

! Example of an individual message (anonymized)

Dear [Student Name] - [login],
Your grade for Mock Exam 4 is available on Moodle.
Below I'm sending the competencies assessed and a detailed review of your code, automatically generated by Artificial Intelligence (AI).
The "rubrica.txt" attachment contains the AI's full feedback (I recommend downloading it and opening it with a text editor or notepad).
Please note that this AI-generated review MAY contain inaccuracies or errors, but it serves as excellent support for your learning process in the course.
A pedagogical suggestion is to submit your code (contained in this file), together with the RUBRIC below, to other LLM models for comparison. This can help identify possible discrepancies, in addition to offering different perspectives on your code and on the assessment criteria.
If you have questions or notice any glaring inconsistency in the review shown on Moodle, I am available for clarification.
Best regards,
Prof. Francisco Zampirolli
PS.: Explaining the process: the latest version of your code saved on Moodle was attached to the prompt and sent to one of the LLMs chosen in an integrated way by our assessment system (models = ("deepseek-chat")). The process is repeated until a response with a valid grade is obtained (between 0 and 100).

Three didactic elements deserve attention in this text: (i) the explicit warning that the review **may contain errors**; (ii) the invitation for the student to **contrast the assessment with other LLMs**, turning the AI into an active study tool rather than a passive verdict; and (iii) the transparent explanation of the automated process, including the model(s) used and the reprocessing policy until a valid grade is obtained.

E.8 Risks, ethical limits, and instructor responsibility

! The instructor cannot simply adopt the AI's grade

This is the central point of this appendix: **under no circumstances should the grade suggested by the AI be automatically assigned to the student as the official grade**. Language models can hallucinate — award points for criteria not met, "invent" that a function was called correctly when it was not, or penalize correct code due to misinterpretation of the rubric. The final grade for any assessment activity must **always** result from the instructor's manual review (or from objective grading via test cases, as in VPL/MCTest), with AI serving, at most, as a **first pass** or as **support material for the student**, never as the decision-making authority.

Some practical precautions adopted in the use of **vpl-ai-feedback**, recommended to any instructor wishing to adapt the system:

- **Clear separation between the official grade and AI feedback.** In the case reported, the mock-exam bonus grade always came from VPL, never from the AI (Section D.1). The AI rubric was communicated to the student as supporting material, with an explicit warning that it may contain errors.
- **Transparency with the student.** The e-mail sent (Section D.6) makes explicit that the assessment was AI-generated, states which model(s) were used, and invites the student to compare with other LLMs — reducing information asymmetry and encouraging critical thinking about the review they received.
- **Reprocessing instead of an invalid *cache*.** The system only writes `rubrica.txt` when it obtains a valid assessment (Section D.4); this prevents a malformed, incomplete, or visibly inconsistent AI response from being presented to the student as if it were final.
- **Monitoring discrepancies.** The consolidated CSV report (the `Diferenca = Total_IA - Total_Moodle` column) allows the instructor to quickly identify the cases where AI and objective grading disagree the most — as in the example in Section D.5 (+20 points) —, prioritizing those cases for manual review.
- **Open channel for disputes.** The final e-mail explicitly offers the student the option to report inconsistencies, keeping the instructor as the final authority over the grade.
- **Personal data kept out of the *prompt*.** As described in Section D.2, name, e-mail, login, student ID, and national ID are never sent to the LLM API, reducing privacy risks when using external AI providers.

E.9 Final remarks

vpl-ai-feedback shows how AI can enrich the feedback cycle in programming courses with a high volume of submissions, offering each student a qualitative, individualized reading of their own code — something unfeasible to do manually for a hundred students, across multiple questions and exams throughout the term. The opinion survey cited in Section D.1 suggests that this type of feedback is well received by students. Even so, responsible use of the system depends on a non-negotiable principle, reiterated throughout this appendix: **AI complements, but does not replace, the instructor's assessment**, and the official grade for any assessment activity must continue to be defined by objective grading (VPL/MCTest, Appendices A and B) and/or human judgment — never by the raw grade returned by a language model.

THE JOURNEY OF DIP & CV

SYSTEMATIZING KNOWLEDGE, FROM PIXEL TO INTELLIGENCE

Part I — Foundations of DIP

- 1. Fundamentals and First Steps
- 2. Sampling, Quantization, and Connectivity
- 3. Spatial Operations and Filtering
- 4. Mathematical Morphology and Segmentation
- 5. Transforms and Compression

Part II — Computer Vision

- 6. Document Analysis and Inspection
- 7. Classification and Pattern Recognition
- 8. Scene Understanding and Segmentation
- 9. Deep Learning (Deep Learning)

Essential Appendices

- **A:** MCTest
- **B:** Moodle/VPL
- **C:** SEB/Simulators
- **D:** Gemini/Teaching Material
- **E:** AI in the vpl-ai-feedback
- **F:** Perception and Assessment

INPUT IMAGE (PIXEL) → **DIP** (PROCESSING) → **CV** (VISION) → INTELLIGENT OUTPUT (SEMANTIC)

MASTER THE PIXELS, BUILD THE FUTURE.

From simple histograms to complex scene understanding with deep learning, this work offers a complete roadmap. Transform images into data, and data into intelligent decisions.

Your ability to shape computational perception starts here. Stay ahead, explore, and innovate!

Available in digital format (HTML, PDF, and Jupyter Notebook), with simulators and exercises, including test cases with VPL and Moodle.

https://fzampirolli.github.io/dip-cv/

SCAN AND ACCESS EXTRA CONTENT