PDI & VC

Procesamiento Digital de Imágenes y Visión por Computadora

Fundamentos, Algoritmos y Aplicaciones Integradas

AUTOR: FRANCISCO DE ASSIS ZAMPIROLLI

Procesamiento Digital de Imágenes y Visión por Computador

Libro interactivo con Python

Francisco de Assis Zampirolli

Universidade Federal do ABC

20 de septiembre de 2026

© 2026 Francisco de Assis Zampirolli da Universidade Federal do ABC (UFABC).
Todos os direitos reservados.

Este livro está sob a Licença:
Creative Commons Attribution-ShareAlike 4.0 International License
Detalhes no endereço: creativecommons.org/licenses/by-sa/4.0

Projeto gráfico: Francisco de Assis Zampirolli
Capa: Francisco de Assis Zampirolli, com suporte de IA (Gemini e ChatGPT)

CATALOGAÇÃO NA FONTE
SISTEMA DE BIBLIOTECAS DA UNIVERSIDADE FEDERAL DO ABC

[CÓDIGO CUTTER] Zampirolli, Francisco de Assis
Processamento Digital de Imagens e Visão Computacional / Francisco de Assis Zampirolli – Santo André, SP : Edição do Autor, 2026.

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

ISBN: [ISBN A DEFINIR] (1ª Edição)

1. Processamento Digital de Imagens. 2. Visão Computacional. 3. Morfologia Matemática. 4. Python. 5. Correção Automática. 6. Moodle. I. Título.

CDD [XX] ed. – [CÓDIGO CDD]

Índice

PDI y VC — Python 1
Organización . . . 1

Prefacio 3
Contexto de la primera edición . . . 3
Cómo se produce este libro . . . 4
Uso de herramientas de Inteligencia Artificial . . . 5
La biblioteca `morph.py` . . . 6
Ejercicios de Programación (EPs) y Validación Automática . . . 7
Código abierto . . . 7
Antes de comenzar: *Notebooks* en Python . . . 8
Guía del Profesor: Publicación de EPs en Moodle . . . 8
Integración con Moodle (VPL) . . . 8
Cómo Publicar en Moodle . . . 9

I Parte I — Fundamentos de PDI 1

1 Fundamentos y Primeros Pasos 3
1.1 Objetivos . . . 3
1.2 Antes de comenzar: *Notebooks* interactivos . . . 3
1.3 Fundamentos . . . 3
1.3.1 👁️ Visión por Computador . . . 4
1.3.2 🖼️ Procesamiento Digital de Imágenes (PDI) . . . 4
1.4 Etapas del PDI . . . 5
1.5 Formación de la Imagen y el Espectro . . . 5
1.6 ¿Qué es una Imagen Digital? . . . 7
Representación Matemática . . . 7
1.7 Configuración del Entorno . . . 9
1.8 Versiones de `morph.py` . . . 9
1.9 Fundamentos de Matrices — atención a la copia de referencias . . . 10
1.10 Leyendo y Mostrando Imágenes . . . 11
Alternativa: *Descarga* de la Imagen para Almacenamiento Local . . . 12
1.11 Conversión de Tipos y Umbralización . . . 13
Conversión a Escala de Grises . . . 13
Umbralización (*Thresholding*) . . . 13
Ejemplo práctico de conversión y umbralización . . . 14
1.12 Umbralización por el método de Otsu . . . 14
1.13 Acceso a Píxeles . . . 15
1.14 Resumen . . . 17
1.15 🤖 Uso del Gemini Notebook como Tutor Complementario . . . 18
Funcionalidades Disponibles en la Plataforma . . . 18
1.16 Lista de Ejercicios . . . 19
Referencias del Capítulo . . . 19
1.17 💻 **Parte Práctica con Ejercicios de Programación** . . . 19
🎯 Objetivo de este Cuaderno . . . 19

§ Instrucciones Paso a Paso . . . 20
⚠️ Importante: Reglas y Buenas Prácticas . . . 21
1.17.1 EP01_01 📏 Tres métricas de distancia en PDI . . . 21
1.17.2 EP01_02 📊 Desempeño Predictivo — Métricas de ML en VC . . . 29
1.17.3 EP01_03 ↗ *Mean Average Precision* (mAP) — Curva Precisión-Sensibilidad . . . 30
1.17.4 EP01_04 🖼️ Lectura e Información de una Imagen en Matriz . . . 33
1.17.5 EP01_05 🔄 Negativo de una Imagen en Tonos de Gris . . . 36
1.17.6 EP01_06 🎨 Conversión RGB → Escala de Grises (ITU-R BT.601) . . . 37
1.17.7 EP01_07 ⚫ Umbralización Manual: Imagen Binaria . . . 38
1.17.8 EP01_08 🎨 Remapeamento por Rango de Intensidad . . . 40
1.17.9 EP01_09 🏁 Patrón de Tablero: Matriz de Ajedrez . . . 42
1.17.10 EP01_10 📄 Metadatos: Lectura de Archivo PGM . . . 44
1.17.11 EP01_11 ↗ Análisis de Vecindad: Filtro de Máximo 1D . . . 46

2 De la Captura al Píxel - Muestreo, Cuantización y Conectividad **49**
2.1 Objetivos . . . 49
2.2 El Ojo y la Cámara - Elementos de la Percepción Visual . . . 49
2.2.1 Visión humana . . . 49
2.2.2 Sensores digitales . . . 49
2.3 Ilusiones Ópticas: Los Desafíos de la Percepción Visual . . . 50
2.3.1 Ambigüedad y Contexto . . . 50
2.3.2 Brillo y Contraste Local . . . 51
2.3.3 Geometría y Perspectiva . . . 51
2.4 El Modelo Matemático de la Formación de la Imagen . . . 51
2.4.1 Representación Digital y Canales Espectrales . . . 52
2.4.2 Preparando el Entorno Práctico . . . 53
2.4.3 Implementación de la Lectura de Imagen en `morph.py` . . . 53
2.4.4 RGB y RGBA: Ejemplo Práctico con la Imagen Mandrill . . . 54
2.5 Digitalización: Muestreo y Cuantización . . . 55
2.5.1 Muestreo - Discretización del espacio . . . 55
2.5.2 Cuantización - Discretización de la intensidad . . . 55
2.5.3 Efectos del muestreo y la cuantización . . . 56
2.5.4 Análisis Técnico . . . 58
2.6 Relaciones entre Píxeles - Topología de la Imagen . . . 59
2.6.1 Vecindad . . . 59
2.6.2 Adyacencia, conectividad y caminos . . . 59
2.6.3 Distancias entre píxeles . . . 60
2.7 Almacenamiento de Imágenes . . . 60
2.7.1 Ejemplo: Extracción de Metadatos y Localización GPS . . . 61
2.7.2 ¿Por qué es importante esta separación? . . . 63
2.8 Transformaciones Geométricas Básicas . . . 63
2.8.1 Traslación . . . 64
2.8.2 Rotación . . . 64
2.8.3 Escala (Redimensionamiento) . . . 65
2.8.4 Cisallamiento (*Shear*) . . . 66
2.9 Resumen . . . 67
2.10 🤖 Uso del Gemini Notebook como Tutor Complementario . . . 68
2.11 Lista de Ejercicios . . . 68
Referencias del Capítulo . . . 68
2.12 💻 **Parte Práctica con Ejercicios de Programación** . . . 69
🎯 Objetivo de este Cuaderno . . . 69
2.12.1 EP02_01 ⊙ Ajuste de Brillo y Contraste . . . 69
2.12.2 EP02_02 🔬 Submuestreo Espacial . . . 71
2.12.3 EP02_03 🎨 Cuantización de Niveles de Gris . . . 73

2.12.4 EP02_04 📐 Transformada de Distancia en Imagen Binaria . . . 75
2.12.5 EP02_05 Transposición de Imagen . . . 77
2.12.6 EP02_06 🔄 Rotación de Imagen . . . 78
2.12.7 EP02_07 🔍 Redimensionamiento (Escala) . . . 80
2.12.8 EP02_08 🔀 Cortante (Shear) . . . 84
2.12.9 EP02_09 🧩 Transformación Afín Genérica . . . 85
2.12.10 EP02_10 🎯 Corrección de Perspectiva (Homografía) . . . 87
2.12.11 EP02_11 🏆 Corrección de Perspectiva (Homografía) en Imagen Real . . . 91

3 Operaciones Espaciales: Intensidad, Histograma y Filtrado 97
3.1 Objetivos . . . 97
3.2 Operaciones a Nivel de Intensidad . . . 97
3.2.1 Preparando el Entorno Práctico . . . 98
3.2.2 Operaciones Aritméticas . . . 99
3.2.3 Mezcla Ponderada (*Alpha Blending*) . . . 100
3.2.4 Operaciones Lógicas y Máscaras Bit a Bit . . . 101
3.3 Histograma de Imágenes . . . 102
3.3.1 Ecualización de Histograma . . . 103
3.3.2 Especificación de Histograma . . . 107
3.4 Fundamentos Espaciales: Vecindad, Convolución y *Kernels* . . . 108
3.4.1 Vecindad . . . 109
3.4.2 Tratamiento de Bordes . . . 109
3.4.3 Correlación vs. Convolución . . . 110
3.4.4 El Papel del *Kernel* . . . 112
3.4.5 Ejemplo Numérico: Correlación Paso a Paso . . . 115
3.5 Filtrado Espacial de Suavizado . . . 116
3.5.1 Filtro de Media (*Box Filter*) . . . 116
3.5.2 Filtro Gaussiano . . . 116
3.6 Filtrado Espacial de Realce . . . 119
3.6.1 Laplaciano . . . 119
3.6.2 Operador de Sobel . . . 123
3.6.3 Operador de Prewitt . . . 125
3.6.4 *Unsharp Masking* (USM) . . . 126
3.6.5 Detector de Canny . . . 128
3.7 Filtros de Orden: Filtro de la Mediana . . . 129
3.7.1 Ruido Impulsivo: Sal y Pimienta . . . 129
3.7.2 Filtro de la Mediana . . . 130
3.8 Aplicación Práctica: Preprocesamiento para Segmentación . . . 133
3.9 Resumen . . . 134
3.10 🤖 Uso del Gemini Notebook como Tutor Complementario . . . 135
3.11 Lista de Ejercicios . . . 135
Referencias del Capítulo . . . 136
3.12 💻 **Parte Práctica con Ejercicios de Programación** . . . 136
🎯 Objetivo de este Cuaderno . . . 136
3.12.1 EP03_01 ➕ Adición Saturada de Constante . . . 137
3.12.2 EP03_02 🔀 Alpha *Blending* de Dos Imágenes . . . 139
3.12.3 EP03_03 🎭 Inversión de Imagen (Negativo Fotográfico) . . . 140
3.12.4 EP03_04 📊 Equalización de Histograma (L bits) . . . 142
3.12.5 EP03_05 🔲 Aplicación de Máscara AND Binaria . . . 143
3.12.6 EP03_06 Filtro de Media con *Kernel* N×N . . . 146
3.12.7 EP03_07 🔍 Operador Laplaciano (w4) para Realce de Bordas . . . 149
3.12.8 EP03_08 🧭 Gradiente de Sobel: Gx y Gy . . . 150
3.12.9 EP03_09 📡 Filtro de la Mediana 3×3 . . . 152
3.12.10 EP03_10 ✨ *Unsharp Masking* (USM) . . . 156

4 Morfología Matemática y Segmentación de Imágenes **159**
4.1 Objetivos . . . 159
4.2 Umbralización . . . 160
4.2.1 Imagen de Monedas . . . 161
4.2.2 Preprocesamiento para el Otsu . . . 161
4.2.3 Resultado: CLAHE como Mejor Preprocesamiento . . . 164
4.3 Morfología Matemática . . . 165
4.3.1 Erosión y Dilatación . . . 165
4.3.2 Apertura y Cierre . . . 173
4.3.3 Operadores Geodésicos . . . 175
4.3.4 Reconstrucción Morfológica . . . 179
4.3.5 Relleno de Agujeros y Eliminación de Bordes . . . 181
4.3.6 *Pipeline* de Limpieza Binaria con CLAHE . . . 184
4.3.7 Morfología en Tonos de Gris . . . 185
4.4 Segmentación de Imágenes: Fundamentación y Taxonomía . . . 188
4.4.1 Etiquetado . . . 189
4.4.2 Transformada de Distancia . . . 193
4.4.3 Transformada de Distancia Euclidiana en cuatro pasos . . . 196
4.4.4 Segmentación por *Watershed* . . . 198
4.5 Extracción de Componentes y Descriptores de Forma . . . 204
4.5.1 Rotulado y Estadísticas de Componentes . . . 205
4.5.2 Descriptores de Forma . . . 206
4.5.3 Conexión con la Detección de Objetos Moderna . . . 208
4.6 Resumen . . . 210
4.7 🤖 Uso de Gemini Notebook como Tutor Complementario . . . 211
4.8 Lista de Ejercicios . . . 212
Referencias del Capítulo . . . 212
4.9 EP4: Adición de Ruido y Filtros . . . 213
4.10 Cómo usar este notebook . . . 213
4.11 Bibliotecas necesarias . . . 213
4.12 Función para mostrar imágenes . . . 213
4.13 Funciones para agregar ruido . . . 213
4.13.1 Ruido sal y pimienta . . . 213
4.13.2 Ruido gaussiano . . . 214
4.13.3 Ruido uniforme . . . 214
4.14 Funciones para filtros . . . 215
4.14.1 Filtro de media . . . 215
4.14.2 Filtro de mediana . . . 215
4.14.3 Filtro gaussiano . . . 216
4.15 Carga de la imagen . . . 216
4.16 Aplicación de ruido . . . 216
4.17 Aplicación de filtros sobre imágenes ruidosas . . . 217
4.17.1 Filtro de media sobre diferentes ruidos . . . 217
4.17.2 Filtro de mediana sobre diferentes ruidos . . . 217
4.17.3 Filtro gaussiano sobre diferentes ruidos . . . 217
4.18 Comparación de resultados . . . 218
4.19 Ejercicio de análisis . . . 218
4.20 💻 **Parte Práctica con Ejercicios de Programación** . . . 218
🎯 Objetivo de este cuaderno . . . 219
4.20.1 EP04_01 Umbralización Global por Umbral Fijo . . . 219
4.20.2 EP04_02 📊 Umbralización Automática de Otsu . . . 221
4.20.3 EP04_03 🌱 Dilatación Binaria Plana (mm.dil0) . . . 223
4.20.4 EP04_04 🪨 Erosión Binaria Plana (mm.ero0) . . . 226

4.20.5 EP04_05 🧹 Apertura Morfológica (Eliminación de Ruido) . . . 227
4.20.6 EP04_06 🧩 Cierre Morfológico (Relleno de Huecos) . . . 231
4.20.7 EP04_07 Dilatación y Erosión con Pesos (mm.dil1 / mm.ero1) . . . 233
4.20.8 EP04_08 🌋 Gradiente morfológico, Top-hat y Black-hat . . . 234
4.20.9 EP04_09 Transformada de Distancia y el "Centro" del Objeto . . . 237
4.20.10 EP04_10 🪙 Separación de *Blobs*, Etiquetado y Descriptores . . . 239

5 Transformadas y Compresión 243
5.1 Objetivos . . . 243
5.2 Configuración del Entorno . . . 244
5.3 Transformada de Fourier Discreta 2D . . . 244
5.3.1 Simulador: Reconstruyendo Señales con Senoides . . . 244
5.3.2 Interpretación del espectro de frecuencia . . . 246
5.3.3 El Experimento de la Rejilla: Construyendo una Imagen a partir de un Único Coeficiente 246
5.3.4 Definición Matemática . . . 248
5.3.5 Magnitud y Fase . . . 248
5.3.6 ¿Qué transportan la magnitud y la fase? . . . 250
5.4 Teorema de la Convolución y Estrategias de Filtrado . . . 251
5.4.1 *Kernel* Separable vs. No separable . . . 251
5.4.2 Análisis de Eficiencia Computacional . . . 252
5.4.3 Discusión de los resultados experimentales . . . 252
5.5 Filtros en el dominio de la frecuencia . . . 256
5.5.1 Filtros Pasa-Bajos . . . 257
5.5.2 Filtros Pasa-Alto y Pasa-Banda . . . 258
5.5.3 Eliminación de Ruido Periódico . . . 262
📌 Síntesis — Filtros Espectrales . . . 264
5.6 *Wavelets* y Multirresolución . . . 264
5.6.1 El Límite de la Transformada de Fourier: localización espacial . . . 264
5.6.2 Transformada Wavelet Discreta 2D . . . 265
5.6.3 Familias de *Wavelets* . . . 265
5.6.4 Análisis Multirresolución con la DWT 2D . . . 268
Síntesis — Fourier vs. *Wavelets*: ¿cuándo utilizar cada enfoque? . . . 274
5.7 Compresión de Imágenes . . . 275
5.7.1 Taxonomía de las Redundancias . . . 275
5.7.2 Transformada de Cosenos Discreta (DCT-II 2D) . . . 276
5.7.3 Las Funciones de Base de la DCT . . . 276
5.7.4 Concentración de Energía y Reconstrucción Progresiva . . . 277
5.7.5 El *Pipeline* de Compresión JPEG . . . 279
5.7.6 Simulador Interactivo: Cuantización DCT . . . 282
5.8 Comparación de Formatos de Imagen . . . 283
5.8.1 Características de los Formatos . . . 283
5.8.2 Métricas de Evaluación de Calidad . . . 283
5.8.3 Inspección Visual: Naturaleza de los Artefactos de Compresión . . . 284
5.8.4 Evaluación Cuantitativa y Espacial de la Compresión . . . 285
Síntesis — Compresión JPEG . . . 290
5.9 Aplicación Práctica: Eliminación de Ruido mediante Filtrado Híbrido . . . 291
5.9.1 Análisis de Rendimiento y Conclusión del Capítulo . . . 291
5.10 Resumen del Capítulo . . . 293
Fundamentos Esenciales . . . 293
5.11 🤖 Uso de Gemini Notebook como Tutor Complementario . . . 295
5.12 Lista de Ejercicios . . . 295
Referencias del Capítulo . . . 296
5.13 💻 **Parte Práctica con Ejercicios de Programación** . . . 297
🎯 Objetivo de este Cuaderno . . . 297

5.13.1 EP05_01 🟢 Filtro Pasa-Bajas Ideal por Distancia en el Espectro . . . 298
5.13.2 EP05_02 🟡 Filtro *Notch*: Eliminando Picos Periódicos . . . 299
5.13.3 EP05_03 🟠 Cuantización DCT: la Verdadera Fuente de Compresión . . . 301
5.13.4 EP05_04 🔴 Implementando la DFT 2D a partir de la definición . . . 304
5.13.5 EP05_05 🏆 *Pipeline* JPEG Completo: DCT, Cuantización y Reconstrucción . . . 306

II Parte II — Visión por Computador 309

6 Análisis de Documentos e Inspección Industrial 311
6.1 Objetivos del Capítulo . . . 311
6.2 Configuración del Entorno . . . 312
6.3 Bases de Imágenes para Experimentación . . . 313
6.3.1 Imágenes Públicas con `skimage.data` . . . 314
6.4 Normalización de Fondo y Ecualización Local de Contraste . . . 316
6.5 Reconocimiento Óptico de Caracteres (OCR) . . . 318
6.6 Traducción Automática del Texto Reconocido . . . 320
6.7 Fundamentos de OMR e Inspección Industrial . . . 322
6.8 Proyectos Prácticos: Construcción de un *Pipeline* de Análisis Documental . . . 322
6.8.1 Alineación Automática de Documentos (*OCR/OMR Pre-processing*) . . . 323
6.8.2 Algoritmo de Rectificación de Inclinación (*Deskew*) . . . 324
6.8.3 Modelado Matemático . . . 324
6.8.4 Limitaciones Prácticas . . . 328
6.8.5 Detección de Bordes y Contornos . . . 329
6.8.6 Aislamiento, Segmentación y Decodificación del *QRCode* . . . 332
6.8.7 Decodificación de Códigos de Barras . . . 334
6.9 El MCTest como Estudio de Caso: del Prototipo al Sistema en Producción . . . 337
6.9.1 Obtención y Preparación del Módulo . . . 337
6.9.2 Extracción del Área de Respuestas . . . 338
6.9.3 Segmentación del *QRCode* . . . 340
6.9.4 Ubicación de los Cuadros de Respuestas . . . 342
6.9.5 Lectura Automática de las Respuestas . . . 342
6.10 Inspección Industrial Automatizada . . . 344
6.10.1 Referencias y *Datasets* Públicos . . . 345
6.10.2 Detección de Defectos mediante Análisis de Textura . . . 346
6.11 Resumen . . . 349
Próximos Pasos . . . 350
6.12 🤖 Uso de Gemini Notebook como Tutor Complementario . . . 350
6.13 Lista de Ejercicios . . . 351
Referencias del Capítulo . . . 352
6.14 💻 Parte Práctica con Ejercicios de Programación . . . 352
Leyenda de Dificultad . . . 353
🎯 Objetivo de este Cuaderno . . . 353
6.14.1 EP06_01 🟢 Evaluación de Segmentación por IoU (*Intersection over Union*) . . . 354
6.14.2 EP06_02 🟢 Filtro de Marcadores por Circularidad . . . 357
6.14.3 EP06_03 🟡 Clasificación de Marcaciones en Hojas de Respuesta (OMR) . . . 358
6.14.4 EP06_04 🟡 Estimador de Inclinación por Mediana Angular (*Deskew*) . . . 360
6.14.5 EP06_05 🟠 Normalización de Fondo por División (Corrección de Iluminación) . . . 362
6.14.6 EP06_06 🔴 Mapa de Varianza Local para Detección de Textura . . . 364
6.14.7 EP06_07 🟣 *Pipeline* de Inspección Industrial: Registro por Traslación y Sustracción . . . 366
6.14.8 EP06_08 ⚫ Segmentación y Decodificación Real de *QRCode* con OpenCV . . . 369

7 Clasificación de Imágenes y Reconocimiento de Patrones 375
7.1 Objetivos del Capítulo . . . 375

7.2 Configuración del Entorno . . . 375
7.3 Un Problema Concreto: Clasificación de Frutas . . . 377
7.4 Panorama del Capítulo: El *Pipeline* Clásico de Clasificación de Imágenes . . . 379
7.5 Fundamentos del Reconocimiento de Patrones . . . 379
7.6 Extracción de Descriptores Clásicos . . . 380
7.6.1 *Local Binary Patterns* (LBP) . . . 381
7.6.2 *Histograma de Gradientes Orientados* (HOG) . . . 383
7.6.3 El Impacto de la Escala y la Normalización de Características . . . 385
7.7 📌 Mapa Conceptual . . . 387
7.8 Descriptores en la Práctica . . . 387
7.9 Un Problema Concreto: Simulando Descriptores de Frutas . . . 392
7.10 Clasificador k-NN: Cómo Funciona por Dentro . . . 395
7.10.1 Métrica de Distancia . . . 396
7.10.2 Regla de Decisión . . . 396
7.10.3 El Papel del Parámetro k . . . 398
7.11 Proyecto Práctico 1: Clasificación de Dígitos Manuscritos con k-NN . . . 399
7.11.1 Clasificación con Vectores de Intensidad . . . 400
7.11.2 Clasificación con Descriptores HOG . . . 401
7.12 Evaluación de Clasificadores . . . 403
7.12.1 Elección de k por Validación Cruzada . . . 403
7.12.2 El Compromiso entre Sesgo y Varianza: Diagnóstico de *Overfitting* y *Underfitting* . . . 404
7.13 Proyecto Práctico 2: Comparación de Descriptores para Clasificación de Texturas . . . 406
7.13.1 *Pipeline* de Extracción de Características y Evaluación Comparativa . . . 408
7.13.2 Diagnóstico Fino del Clasificador: Precisión, Exhaustividad y Puntuación F1 . . . 411
7.14 Limitaciones de los Descriptores Artesanales . . . 413
7.15 Resumen . . . 414
7.16 🤖 Uso del Gemini Notebook como Tutor . . . 414
7.17 Lista de Ejercicios . . . 414
Referencias del Capítulo . . . 415
7.18 💻 **Parte Práctica con Ejercicios de Programación** . . . 416
🎯 Objetivo de este Cuaderno . . . 416
Download . . . 416
Resumen de los Métodos de `morph.py` (Cap. 7) . . . 417
7.18.1 EP07_01 🟢 Clasificador k-NN Paso a Paso . . . 417
7.18.2 EP07_02 🟡 Normalización *Z-score* y Robustez del k-NN ante Escalas Distintas . . . 419
7.18.3 EP07_03 🟡 Evaluación por Matriz de Confusión . . . 422
7.18.4 EP07_04 🟠 Codificación Manual del Descriptor LBP . . . 424
7.18.5 EP07_05 🔴 Histograma de Orientaciones de una Célula HOG . . . 425
7.18.6 EP07_06 🟣 *Pipeline* Completo: Descriptores + k-NN + Evaluación Multiclase . . . 428
7.18.7 EP07_07 ⚫ Clasificación Real de un Mosaico de Texturas mediante LBP + k-NN . . . 431

8 Comprendiendo Escenas: Correspondencia de Características, Detección y Segmentación 439
8.1 Objetivos del Capítulo . . . 439
8.2 Configuración del Entorno . . . 440
8.3 Detectores y Descriptores Locales de Características . . . 441
8.3.1 El Algoritmo ORB (*Oriented FAST and Rotated BRIEF*) . . . 441
8.3.2 Explorando el Simulador del ORB . . . 442
8.4 Correspondencia de Características y Homografía con ORB . . . 444
8.4.1 Detectando y Correspondiendo Características con ORB . . . 445
8.5 Modelagem Matemática: Homografía y RANSAC . . . 447
8.5.1 Homografía . . . 447
8.5.2 RANSAC . . . 447
8.5.3 Estimando la Homografía con RANSAC . . . 449
8.5.4 Registro de la Imagen . . . 450

8.6 Detección de Objetos: *Haar Cascade* (Viola-Jones) . . . 452
8.7 Detecção de Objetos: Caixas Delimitadoras, IoU e NMS . . . 454
8.7.1 *Intersection over Union* (IoU) . . . 455
8.7.2 Supresión de No-Máximos (NMS) . . . 455
8.8 Segmentación Visual: Semántica, de Instancias y Panóptica . . . 457
8.8.1 Segmentación Semántica por Umbralización . . . 457
8.8.2 Segmentación de Instancias por Etiquetado . . . 457
8.8.3 Alternativa: Transformada de Distancia + *Watershed* . . . 459
8.8.4 Medición y Extracción de Atributos con `mm.measure` . . . 461
8.8.5 Exportación de Anotaciones: de `mm.measure` al Formato YOLO . . . 463
8.8.6 Validação de Anotaciones por IoU: `mm.verifyBoundBox` . . . 464
8.8.7 Visualización de Anotaciones Guardadas: `mm.showBoundBox` . . . 465
8.9 Visión General de los Modelos Modernos . . . 466
8.10 Limitaciones de los Enfoques Clásicos y Motivación para el *Deep Learning* . . . 467
8.11 Resumen . . . 467
Próximos Pasos . . . 468
8.12 🤖 Uso de Gemini Notebook como Apoyo al Estudio . . . 468
8.13 Lista de Ejercicios . . . 468
Referencias del Capítulo . . . 469
8.14 💻 **Parte Práctica con Ejercicios de Programación** . . . 470
🎯 Objetivo de este Cuaderno . . . 470
Download . . . 470
Resumen de los Métodos de `morph.py` (Cap. 8) . . . 471
8.14.1 EP08_01 🟢 Distancia de Hamming y Correspondencia de Descriptores Binarios . . . 471
8.14.2 EP08_02 🟢 Homografía y RANSAC: La Votación por *Inliers* . . . 473
8.14.3 EP08_03 🟢 Imagen Integral: Sumas Rectangulares en Tiempo Constante . . . 475
8.14.4 EP08_04 🟢 IoU y Supresión de No Máximos (NMS) . . . 477
8.14.5 EP08_05 🟡 Etiquetado de Componentes Conectados: Segmentación de Instancias . . . 479
8.14.6 EP08_06 🟡 *Bounding Boxes*, Centroides y Propiedades de Instancias con `mm.measure` 481
8.14.7 EP08_07 🟡 Eliminación de Ruido Sal y Pimienta y Medición de Objetos . . . 484
8.14.8 EP08_08 🟡 Imagen en Niveles de Gris y Umbralización Dinámica . . . 485
8.14.9 EP08_09 🟠 Gradiente de Iluminación y Umbralización Adaptativa . . . 488
8.14.10 EP08_10 🔴 Contraste Bajo y Separación de Objetos Tópicos (*Watershed* / Distancia) 490
8.14.11 EP08_11 🔴 Clasificación y Validación de Objetos con Plantilla de *Bounding Box* . . . 492
8.14.12 EP08_12 🔴 Segmentación de Instancias en Imagen Real: Objetos Geométricos . . . 495

9 Aprendizaje Profundo para Visión por Computador **499**
9.1 Objetivos del Capítulo . . . 499
9.2 Panorama: Clasificación, Detección y Segmentación . . . 500
9.3 Configuración del Entorno . . . 501
9.4 Fundamentos del Aprendizaje Profundo para VC . . . 503
9.4.1 De la Convolución Fija a la Convolución Aprendida . . . 503
9.4.2 Neurona Artificial . . . 503
9.4.3 Capa Convolucional . . . 504
9.4.4 Función de Activación . . . 506
9.4.5 *Pooling* . . . 506
9.4.6 Entrenamiento de Redes Neuronales: Cómo Aprenden las CNNs . . . 508
9.4.7 Arquitectura de una CNN . . . 511
9.4.8 Cómo el Gradiente Ajusta los *Kernels* de la Convolución . . . 511
9.5 Aplicaciones Prácticas en VC . . . 518
9.5.1 Clasificación de Imágenes con CNNs . . . 518
9.5.2 Detección de Objetos . . . 548
9.5.3 Segmentación de Objetos . . . 559
9.5.4 Ingeniería de Datos para VC (*Roboflow*) . . . 569

9.5.5 Aplicaciones Geométricas e Integradas . . . 574
9.6 Resumen . . . 578
9.7 🤖 Uso de Gemini Notebook como Tutor Complementario . . . 579
9.8 Lista de Ejercicios . . . 580
9.9 Cierre de la Parte II . . . 581
Referencias del Capítulo . . . 582
9.10 💻 **Parte Práctica con Ejercicios de Programación** . . . 582
🎯 Objetivo de este Cuaderno . . . 583
9.10.1 EP09_01 🟢 Convolución 2D Manual (*Forward* de una Capa Aprendida) . . . 584
9.10.2 EP09_02 🟢 *Pooling* Manual (Máximo y Media) . . . 587
9.10.3 EP09_03 🟡 Conteo de Parámetros Entrenables de una CNN . . . 590
9.10.4 EP09_04 🟡 Intersección sobre Unión (IoU) y Supresión de No-Máximos (NMS) . . . 593
9.10.5 EP09_05 🟠 Evaluación de Segmentación: IoU y Dice Pixel a Pixel . . . 596
9.10.6 EP09_06 🔴 *Pipeline* Integrado: De la Detección a la Medición del Mundo Real . . . 601

Referencias **605**

Apéndices **607**

A Sobre o MCTest **607**
A.1 Instalação do MCTest . . . 607
A.2 Criando um exame no MCTest . . . 607
A.3 Criando uma questão paramétrica . . . 608
A.4 Exemplo real: uma questão do Simulado 4 . . . 610
A.5 Considerações finais . . . 612

B Sobre o Moodle (VPL) **613**
B.1 Configurando uma atividade VPL . . . 613
B.2 Publicando os EPs como atividades VPL . . . 613
B.3 Considerações finais . . . 614

C Uso do SEB nos simulados quinzenais e avaliações **615**
C.1 Configurando a aplicação no *SEB Tool* . . . 615
C.2 Vinculando a chave do SEB à atividade VPL no Moodle . . . 616
C.3 Restrição por endereço IP (opcional) . . . 616
C.4 Considerações finais . . . 616

D Geração de material didático com o *Gemini Notebook* **617**
D.1 Vídeos conceituais e de resolução de EPs . . . 617
D.2 *Slides* de apoio . . . 617
D.3 Material principal e considerações finais . . . 617

E Uso de IA na Avaliação de Estudantes: o Projeto vpl-ai-feedback **619**
E.1 Contexto de uso . . . 619
E.2 Arquitetura do projeto . . . 620
E.3 O *prompt* universal e a rubrica em duas etapas . . . 621
E.4 Duas camadas adicionais de evidência . . . 622
E.5 Fluxo de execução . . . 623
E.6 Exemplo ilustrativo: rubrica de uma prova com três questões . . . 623
E.7 Envio do *feedback* por e-mail . . . 626
E.8 Riscos, limites éticos e responsabilidade docente . . . 627
E.9 Considerações finais . . . 627

Lista de Figuras

1 Botón clicable **Ejecutar en Colab**, disponible al inicio de las partes **Teórica** y **Práctica** de cada capítulo, que permite la ejecución interactiva de los ejemplos y ejercicios. En la versión PDF, existe un *enlace* HTML directo entre la imagen del simulador y su leyenda. 5

1.1 Diagrama relacional que detalla las distinciones fundamentales, sinergias e interconexiones entre PDI y VC en el contexto de sistemas basados en imágenes. 4
1.2 Representación del flujo secuencial de procesamiento: la salida de cada nivel se convierte en la entrada del nivel subsiguiente. 6
1.3 Flujo detallado del PDI: desde la adquisición sensorial hasta la extracción de atributos y el reconocimiento automatizado, incluyendo el procesamiento en color y la compresión. 6
1.4 Representación del espectro visible y su posición en relación con las demás radiaciones electromagnéticas, destacando la variación de las longitudes de onda de 380 nm a 750 nm. . . 7
1.5 (A) Espectro electromagnético completo en escala logarítmica, con énfasis en el rango visible. (B) Detalle de la luz visible (380-750 nm) y sus colores. (C) Descomposición de la luz blanca por el prisma: la menor longitud de onda sufre mayor refracción, separando UV, visible e infrarrojo. 8
1.6 Ejemplos de imágenes sintéticas representadas matricialmente. 11
1.7 Mandril (Mandrillus sphinx) en ambiente natural. Crédito: Julien Renoult (CC BY 4.0). . . . 12
1.8 Procesamiento básico de imágenes: (a) imagen original, (b) imagen en escala de grises, (c) imagen binarizada por umbral (T=128). 14
1.9 Comparación entre umbralización manual (T=128) y automática (Otsu) sobre la imagen en escala de grises. 15
1.10 Acceso a un píxel específico (r, c) y la representación de su vecindario en la imagen. 17
1.11 Simulador EP01_01: Distancias Euclidiana, City-block y Chessboard 23
1.12 Simulador EP01_02: Rendimiento Predictivo — Métricas de ML 30
1.13 Simulador EP01_03: Precisión Media Promedio (mAP) y Curva P-S 34
1.14 Simulador EP01_04: Estadísticas de Píxeles en Matriz Discreta 5x5 35
1.15 Simulador EP01_05: Transformación de Negativo de Imagen (Inversión de Intensidad) 37
1.16 Simulador EP01_06: Conversión RGB a Escala de Grises (Ponderación Perceptual ITU-R BT.601) 39
1.17 Simulador EP01_07: Umbralización Global Interactiva (Binarización p > T) 40
1.18 Simulador EP01_08: Remapeo por Rango Condicional (Brillo y Contraste por Umbral) . . . 42
1.19 Simulador EP01_09: Generador de Patrón de Cuadros (Lógica de Paridad (i + j) % 2) 43
1.20 Simulador EP01_10: Estructura de Archivo PGM (Encabezado ASCII P2 y Mapeo a Tupla Python) 45
1.21 Simulador EP01_11: Filtro de Máximo Local 1D (Vecindad 1x3 con Condición de Borde) . . 47

2.1 Comparativo didáctico entre el sistema visual biológico y el electrónico: en la parte superior, la anatomía del ojo humano destacando la retina y los fotorreceptores (conos y bastones); debajo, la estructura de una cámara digital detallando el sensor CMOS, la matriz de filtros de Bayer (RGGB) y el proceso de cuantización digital realizado por el ADC. 50
2.2 Colección de desafíos perceptivos: (superior izquierda) Vaso de Rubin — ambigüedad figura-fondo; (superior central) Elefante de Shepard — incongruencia geométrica; (superior derecha) Sombra de Adelson — constancia de brillo basada en el contexto; (inferior izquierda) Cuadrícula de puntos — inhibición lateral; (inferior derecha) Escalera de Schröder. 51

2.3 Imagen RGB (3 canales, *MimesNimes*3) y versión RGBA (4 canales, *MimesNimes*4) con transparencia alfa gradual de izquierda a derecha. Imagen Mandrill — USC SIPI Image Database (dominio público). . . . 55
2.4 Efecto del submuestreo. Los títulos muestran las dimensiones (W x H) y el tamaño de la matriz en memoria (KB). . . . 57
2.5 Efecto de la reducción de la profundidad de bits. Los títulos muestran la cantidad de bits, niveles y el tamaño en memoria (KB). . . . 58
2.6 Ilustración de vecindades 4 en una matriz 3x3. En el centro (1,1), el píxel de interés. . . . 59
2.7 Área de Protección Ambiental Quilombos do Médio Ribeira - Barbudo-rajado (Malacoptila striata). Crédito: Thomas Fuhrmann (CC BY-SA 4.0). . . . 62
2.8 Ejemplo de traslación de la imagen del pájaro con desplazamientos (50,50) y (100,50). . . . 64
2.9 Ejemplo de rotación de la imagen del pájaro en 30° y 45° usando interpolación bilineal. . . . 65
2.10 Comparación de interpolación con zoom en el detalle del ojo (recorte 60×60, ampliado 4×). Observe el efecto de bloques en el vecino más cercano vs. la suavización en la bilineal. . . . 66
2.11 Ejemplo de cizallamiento de la imagen del pájaro: horizontal (shx=0.3), vertical (shy=0.3) y combinado (shx=0.2, shy=0.2). . . . 67
2.12 Simulador EP02_01: Ajuste de Brilho y Contraste Lineal (p' = p +) . . . 71
2.13 Simulador EP02_02: Submuestreo Espacial (Reducción de Resolución por Salto f) . . . 73
2.14 Simulador EP02_03: Cuantización y Profundidad de Bits (Reducción del Número de Niveles de Gris) . . . 75
2.15 Simulador EP02_04: Transformada de Distância em Imagem Binária (Chessboard, City-block y Euclidiana) . . . 77
2.16 Simulador EP02_05: Traslación Geométrica de Imagen (Desplazamiento tx y ty con Relleno de Borde) . . . 79
2.17 Simulador EP02_06: Rotación de Imagen en Torno al Origen por Ángulo . . . 81
2.18 Simulador EP02_07: Redimensionamiento Espacial e Interpolación (Vecino más cercano vs Bilineal) . . . 83
2.19 Simulador EP02_08: Transformación Geométrica de Cortante 2D (Cizallamiento Horizontal y Vertical) . . . 86
2.20 Simulador EP02_09: Transformación Afín 2D (Matriz 2×3) . . . 88
2.21 Simulador EP02_10: Corrección de Perspectiva (Transformación de Homografía 3×3) . . . 90
2.22 Adquisición de la imagen de un Sudoku a la izquierda. A la derecha, conversión a tonos de gris y redimensionamiento. Crédito: Héctor Rodríguez de Guardamar, España (CC BY 2.0). . . 92
2.23 Corrección de perspectiva: original y vista frontal rectificada. . . . 93
2.24 Simulador EP02_11: Corrección de Perspectiva del Sudoku (Homografía 3×3 con Remuestreo Bilineal) . . . 94

3.1 *Mandrill* (*Mandrillus sphinx*) fotografiado en ambiente natural en Sudáfrica. Crédito: Carlos Guilherme Rodrigues (CC BY-SA 3.0). . . . 99
3.2 Operaciones aritméticas saturadas: suma de constante (aclarado) y resta de constante (oscurecido con saturación en 0). . . . 100
3.3 Retrato de un leopardo (*Panthera pardus*) en ambiente natural. Crédito: C. Brück (CC BY-SA 4.0). . . . 101
3.4 *Mezcla alfa* entre recortes alineados de mandril y del leopardo (Figura 3.3) para diferentes valores de . En =1 se ve solo el mandril; en =0, solo el leopardo; valores intermedios fusionan las miradas de las dos imágenes proporcionalmente. . . . 101
3.5 Operaciones lógicas bit a bit con máscara circular: AND (aislamiento de la ROI), OR (iluminación de la ROI) y NOT (negativo). . . . 102
3.6 Histogramas de la imagen original, de una versión oscurecida (−80) y de una aclarada (+80). La resta/suma satura en 0 y 255. . . . 103
3.7 Ecualización de histograma en imagen 5×5 con 3 bits (L=8): imagen original con tonos concentrados en {2,3,4}, imagen ecualizada con tonos redistribuidos para {1,5,7}, y los respectivos histogramas evidenciando la dispersión de las frecuencias. . . . 106

3.8 Ecualización de histograma: mm.equalize (global vía CDF) vs. CLAHE (adaptativa con limitación de contraste). Los histogramas revelan la dispersión progresiva de las intensidades. 107
3.9 Especificación de histograma: mandril mapeada al perfil tonal del leopardo. La CDF de salida aproxima la CDF de referencia. 108
3.10 Correlación con *kernel* de media 3×3: versión didáctica (mm.conv0) vs. vectorizada (mm.conv vía cv2.filter2D). La diferencia máxima de 39 ocurre en los bordes. 114
3.11 *Patch* 5×5 extraído de la imagen del leopardo (posición [250:255, 250:255]). La ventana amarilla resalta la vecindad 3×3 centrada en el píxel [1,1] donde se calculará la correlación. 115
3.12 Filtro de media con *kernels* de tamaño creciente (3×3, 7×7, 15×15). El desenfoque de los bordes aumenta con el tamaño del *kernel*. 116
3.13 *Núcleo* Gaussiano 5×5 (=1): pesos normalizados, mayores en el centro y decrecientes radialmente. La cuadrícula facilita la lectura de cada coeficiente. 117
3.14 Comparación entre filtro de media y Gaussiano (*kernel* 9×9, =0). El Gaussiano preserva mejor los bordes, visible en el detalle del rostro. 119
3.15 Simulador: Filtrado Espacial de Realce 1D (Unsharp Masking y High-Boost) 120
3.16 *Kernel* Laplaciano w4 aplicado al *patch* 5×5: la ventana amarilla resalta la vecindad 3×3 donde se calcula el operador de segunda derivada. 122
3.17 Laplaciano aplicado a la imagen del leopardo: respuesta cruda (bordes) con w4 y w8, e imágenes realzadas por la sustracción del Laplaciano. w8 es más sensible a las diagonales. 123
3.18 Operador de Sobel en la imagen del leopardo: Gx detecta bordes verticales, Gy detecta bordes horizontales, y la magnitud | f| combina ambos, revelando todos los bordes independientemente de la dirección. 125
3.19 Operador de Prewitt: Gx, Gy y magnitud — comparable al Sobel, pero sin ponderación Gaussiana perpendicular. 125
3.20 Realce de nitidez por *Unsharp Masking* en la imagen del leopardo: la imagen suavizada se resta de la original para generar la máscara de alta frecuencia, que luego se reintroduce para resaltar bordes y detalles. 127
3.21 *Unsharp Masking* en la imagen del leopardo con =1 y k de 0.5 a 8.0. Para k>2 aparecen halos en los bordes y el ruido de fondo se nota. 128
3.22 Detector de Canny con diferentes pares de umbral: umbrales bajos capturan más bordes (incluso ruido); umbrales altos retienen solo los bordes más fuertes. 129
3.23 Ruido sal y pimienta con densidades crecientes (2%, 5%, 10%). El parámetro prob indica la fracción total de píxeles corrompidos, mitad sal (255) y mitad pimienta (0). 130
3.24 Comparación de filtros para eliminación de ruido sal y pimienta (10%): Gaussiano, Media, Mediana, Bilateral y Morfológico. Fila superior: imágenes completas; fila inferior: recorte de la región de interés. 133
3.25 *Pipeline* de preprocesamiento: CLAHE → Gaussiano → Canny. Cada etapa prepara la imagen para la siguiente, resultando en bordes limpios y bien definidos. 134
3.26 Simulador EP03_01: Adición Saturada de Constante (p' = clip(p + k)) 138
3.27 Simulador EP03_02: Mezcla alfa de dos imágenes (g = · f1 + (1−) · f2) 140
3.28 Simulador EP03_03: Inversión de Imagen — Negativo Fotográfico (p' = 255 − p) 142
3.29 Simulador EP03_04: Ecualización de Histograma (Niveles L = 2^B) 144
3.30 Simulador EP03_05: Aplicación de Máscara AND Binaria 146
3.31 Simulador EP03_06: Filtro de Media con Kernel N×N 148
3.32 Simulador EP03_07: Operador Laplaciano (w4) para Realce de Bordas 151
3.33 Simulador EP03_08: Gradiente de Sobel (Gx y Gy) 153
3.34 Simulador EP03_09: Filtro de la Mediana 3×3 155
3.35 Simulador EP03_10: *Máscara de enfoque* (USM) 158

4.1 Imagen con monedas de varios tipos. Crédito: GAZI.MD.AHAD (CC BY-SA 4.0). 161
4.2 Comparación de los preprocesamientos: imagen | histograma+T* | ²B(T) | Otsu. La mejor separación bimodal indica el umbral más confiable. 164
4.3 Elemento estructurante en forma de cruz (B_{cruz}) utilizado para conectividad-4. 166

4.4 Erosión y dilatación en imagen binaria 10×10 con elemento estructurante 'L' asimétrico 3×3. Validación de la dualidad erosión–dilatación. 172
4.5 Simulador: Erosión Morfológica (A B_L) 172
4.6 Apertura, cierre, composición y filtro secuencial alternado aplicados a la binarización Otsu de las monedas (preprocesadas por realce de contraste). Elemento estructurante: disco 13×13. . 174
4.7 Simulador interactivo de dilatación geodésica: el marcador f (verde) se propaga paso a paso por los caminos libres de la máscara g, sin atravesar paredes. 176
4.8 Resolución de laberinto mediante Operaciones Complementarias: Invirtiendo la máscara y el marcador al inicio del *pipeline*, el flujo se resuelve directamente en el dominio complementario a través de mm.cero y mm.suprec, eliminando reinversiones redundantes. 179
4.9 *Pipeline* de Reconstrucción Morfológica por Dilatación Condicionada: la máscara contiene dos objetos, el marcador aísla solo el núcleo del objeto principal, y las iteraciones reconstruyen su forma exacta hasta la convergencia. 181
4.10 *Pipeline* de relleno de agujeros (*clohole*): el marcador proviene del borde de la imagen, restringido al complemento f^c. La dilatación geodésica reconstruye el fondo externo; tras la complementación, los agujeros internos quedan rellenos. 183
4.11 *Pipeline* de eliminación de estructuras de borde (*edgeoff*): el marcador captura las raíces conectadas a los extremos, la reconstrucción delimita esos elementos y la resta preserva solo los objetos totalmente internos. 184
4.12 *Pipeline* completo de segmentación con CLAHE: Otsu → apertura (r=9) → clohole → apertura (r=33) → edgeoff. 185
4.13 Simulador interactivo avanzado de morfología con elemento estructurante editable y perfil 1D. 186
4.14 Morfología en tonos de gris y sus respectivos histogramas. La erosión reduce intensidades locales, la dilatación las amplía, el gradiente resalta bordes, y los operadores Top-hat y Black-hat evidencian detalles locales brillantes y oscuros. Elemento estructurante: disco de diámetro 19. 188
4.15 Algoritmo de etiquetado por *flood-fill* con pila. 191
4.16 Simulador interactivo de etiquetado de componentes conexas (*connected component labeling*): visualización de la expansión flood-fill, conectividad-4 y conectividad-8. 192
4.17 Efecto de la conectividad en la etiquetación: la imagen binaria 10×10 contiene píxeles diagonalmente adyacentes. Con conectividad-4, estos píxeles forman componentes distintas; con conectividad-8, se fusionan en un único componente. 192
4.18 Algoritmo de la Transformada de Distancia. 194
4.19 Simulador interactivo de la Transformada de Distancia (TD) iterativa mediante erosión en tonos de gris. Los píxeles fuera de la imagen asumen el valor máximo (144), propagando los costos a partir del fondo interno. 195
4.20 Transformada de Distancia en imagen binaria 10×10. Izquierda: original (*foreground* = 255). Centro: mm.dist1 iterativa (erosiones con cruz). Derecha: mm.dist (L2). 196
4.21 Simulador interactivo 2D (4x4) con sincronización estricta de colas de propagación horizontal (b) para obtener la convergencia exacta descrita en el artículo. 197
4.22 Camino geodésico más corto en un laberinto. Las distancias geodésicas se calculan desde la entrada y la salida usando mm.gdist. Los píxeles cuya suma de las dos distancias es igual a la distancia mínima entre los marcadores pertenecen a un camino óptimo. 198
4.23 Algoritmo Didáctico de *Watershed* por Crecimiento de Regiones Limitado por Máscara. . . . 200
4.24 Simulador interactivo del Algoritmo *Watershed* por propagación morfológica. Dibuja la máscara, coloca los marcadores y ajusta el Elemento Estructurante para observar la inundación. Cuando las cuencas se encuentran simultáneamente, el empate se resuelve asumiendo una de las regiones de manera aleatoria. 201
4.25 *Pipeline watershed* delimitado por máscara en imagen binaria 20×20. 202
4.26 *Pipeline Watershed* para separación de monedas superpuestas: desde la máscara binaria dilatada hasta los contornos finales anotados sobre la imagen original. 204
4.27 Componentes conexos extraídos tras el *pipeline* CLAHE → Otsu → limpieza morfológica. Cada objeto se colorea con color distinto y se anota con su área en píxeles. 206

4.28 Contornos extraídos con *findContours*. Cada moneda se anota con su circularidad — valores cercanos a 1 confirman forma circular. 208
4.29 *Bounding boxes* derivadas de los componentes conectados superpuestas a la imagen original. Las anotaciones se exportan en el formato YOLO (*clase cx cy w h*), con coordenadas normalizadas al intervalo [0,1]. 209
4.30 Simulador EP04_01: Umbralización Global por Umbral Fijo (p' = (p > T) ? 255 : 0) 221
4.31 Simulador EP04_02: Limiarización Automática de Otsu (T* = argmax ²_B(T)) 223
4.32 Simulador EP04_03: Dilatación Binaria Plana (g = f B) 225
4.33 Simulador EP04_04: Erosión Binaria Plana (g = f B) 228
4.34 Simulador EP04_05: Apertura Morfológica (g = (f B) B) 230
4.35 Simulador EP04_06: Cierre Morfológico (g = (f B) B) 232
4.36 Simulador EP04_07: Dilatación y Erosión con Pesos (mm.dil1 / mm.ero1) 235
4.37 Simulador EP04_08: Gradiente Morfológico, Top-hat y Black-hat 237
4.38 Simulador EP04_09: Transformada de Distância (Camadas de Erosión) 239
4.39 Simulador EP04_10: Separación de Blobs, Etiquetado y Descriptores 241

5.1 Descomposición de Fourier 1D: una onda cuadrada (línea discontinua) se aproxima mediante la suma de las primeras sinusoides (líneas de colores). Cuantos más términos, mejor la aproximación. 244
5.2 Simulador interactivo de la descomposición de Fourier 1D: visualización de la suma de senoides con diferentes frecuencias, amplitudes y fases. Añada términos y observe la convergencia hacia formas de onda arbitrarias. 245
5.3 Toda frequencia en el espectro (punto aislado) corresponde a una onda sinusoidal 2D rotada en el dominio espacial. 247
5.4 Simulador interactivo de la síntesis de Fourier 2D. Cambie la posición horizontal (u) y vertical (v) del coeficiente en el espectro de frecuencias centrado y observe cómo la distancia respecto al centro determina la frecuencia espacial (grosor) y el ángulo determina la orientación de la onda sinusoidal generada. 247
5.5 Diagrama conceptual del espectro de Fourier 2D centrado. 249
5.6 Experimento de intercambio de fase: Imagen A (monedas) e Imagen B (patrón geométrico) reconstruidas con magnitudes y fases intercambiadas. El resultado muestra que la estructura visual es **mucho más sensible a la fase** que a la magnitud: cuando la fase de B se mantiene, la imagen resultante preserva la organización espacial de B, incluso con la magnitud de A. La magnitud, por su parte, influye principalmente en el contraste y la textura. Observe que la calidad de la reconstrucción no es perfecta — hay artefactos visibles —, evidenciando la interdependencia entre fase y magnitud para una representación fiel de la imagen. 251
5.7 Comparación de eficiencia: Convolución No Separable (Espacial 2D), Separable (Espacial 1D) y vía FFT. 253
5.8 Sin *padding*, un desplazamiento severo hace que la imagen se filtre al lado opuesto (convolución circular). 254
5.9 Verificación del Teorema de la Convolución: la diferencia píxel a píxel entre la convolución espacial (cv2.filter2D) y la multiplicación en frecuencia (FFT) es numéricamente nula — confirmando la equivalencia teórica. 255
5.10 Una Dualidad Peligrosa: El corte abrupto en la Frecuencia (Cilindro) se transforma obligatoriamente en una Sinc espacial. Sus ondulaciones causan el *ringing* fantasma en los bordes de la imagen. 257
5.11 Filtros pasa-baja. 258
5.12 Simulador interactivo de filtros en el dominio de la frecuencia. 259
5.13 Comparación entre filtros pasa-baja: Ideal (D =30), Gaussiano (D =30) y Butterworth (D =30, n=2). Perfiles de H(u,v) a lo largo de una línea central e imágenes filtradas correspondientes. 261
5.14 Filtro pasa-alta Gaussiano. (a) Original; (b) Filtro pasa-alta (D =30) - los bordes de las monedas y el fondo texturizado se resaltan. 262
5.15 Eliminación de ruido periódico mediante filtro *notch* en el dominio de la frecuencia: (a) imagen con ruido senoidal, (b) espectro mostrando los picos del ruido, (c) máscara *notch* centrada en los picos, (d) imagen restaurada. 263

5.16 La transformada de Fourier global es ciega a la posición. Los espectros no dicen dónde están los bordes. . . . 265
5.17 Funciones de la *Wavelet* ( ). Obsérvese cómo decaen rápidamente a cero (soporte compacto), a diferencia de las sinusoides infinitas de Fourier. . . . 267
5.18 Diagrama de la descomposición *wavelet* 2D en dos niveles. . . . 267
5.19 Simulación de la descomposición *wavelet* 2D. . . . 269
5.20 Descomposición *wavelet* 2D de 2 niveles con *wavelet* Haar: subbandas LL, LH, HL, HH en cada nivel. Las subbandas de detalle revelan estructuras orientadas en diferentes escalas utilizando un patrón sintético. . . . 271
5.21 Comparación entre familias de *wavelets*: Haar, db4, sym4 y bior2.2. Subbanda LL (aproximación) y HH (diagonal) para cada elección, ilustrando el compromiso entre compactación y suavidad con base en el patrón sintético. . . . 272
5.22 Reconstrucción *wavelet* con umbralización de coeficientes (*hard thresholding*): a medida que el umbral aumenta, más detalles se ponen a cero, produciendo imágenes progresivamente más suaves. La métrica PSNR cuantifica la pérdida de calidad sobre el patrón sintético. . . . 274
5.23 El Alfabeto Visual del JPEG: Las 64 funciones de base de la DCT-II. El coeficiente DC está en la parte superior izquierda (suave). Al descender y avanzar hacia la derecha, la oscilación espacial aumenta drásticamente. . . . 277
5.24 DCT 2D en bloque 8×8: coeficientes y reconstrucción progresiva. . . . 279
5.25 *Pipeline* JPEG simplificado aplicado a la imagen clásica del *Cameraman*: DCT en bloques 8×8, cuantización con diferentes factores de calidad y reconstrucción vía IDCT. Los artefactos de bloque (*blocking artifacts*) se vuelven visualmente evidentes en factores de calidad reducidos ($Q = 10$ y $Q = 25$). . . . 282
5.26 Simulador interactivo de compresión DCT-JPEG: ajuste el factor de calidad y visualice en tiempo real los coeficientes anulados, el bloque reconstruido y el error de cuantización. . . . 283
5.27 Análisis comparativo de artefactos de compresión bajo factor de calidad reducido ($Q = 10$). A la izquierda, se observa el artefacto de bloque característico de la discretización por DCT en JPEG. A la derecha, se evidencia el efecto de atenuación y suavizado de bordes intrínseco al estándar WebP. . . . 285
5.28 Curva tasa-distorsión: PSNR vs tamaño de archivo para JPEG, WebP y PNG aplicada a la imagen del *Cameraman*. . . . 287
5.29 Análisis espacial de degradación: imágenes reconstruidas y sus respectivos mapas de error absoluto normalizados para diferentes factores de calidad JPEG. . . . 289
5.30 *Pipeline* completo de eliminación de ruido mixto: (1) adición de ruido gaussiano y periódico; (2) identificación de picos de interferencia en el espectro de frecuencias; (3) aplicación de máscara *notch* con atenuación gaussiana suave; (4) post-procesamiento mediante filtro bilateral para la eliminación del ruido estocástico residual. . . . 293
5.31 Mapa conceptual de las transformaciones y propiedades en el dominio de la frecuencia. . . . 294
5.32 Simulador EP05_01: Filtro Pasabajas Ideal en el Espectro . . . 300
5.33 Simulador EP05_02: Filtro Notch . . . 302
5.34 Simulador EP05_03: Cuantización DCT (*round-trip*) . . . 303
5.35 Simulador EP05_04: DFT 2D manual . . . 305
5.36 Simulador EP05_05: *Pipeline* JPEG completo en bloques . . . 307

6.1 Muestra de imágenes públicas de *skimage.data*, agrupadas por área de aplicación. . . . 316
6.2 Comparación entre estrategias de preprocesamiento para binarización de la imagen de texto: (a) imagen original; (b) umbralización directa por el método de Otsu; (c) fondo estimado por filtrado Gaussiano; (d) imagen tras normalización de fondo (división por la imagen suavizada); (e) CLAHE seguido de umbralización por Otsu; y (f) normalización de fondo seguida de umbralización por Otsu. Las imágenes ilustran el efecto de cada técnica en la compensación de variaciones de iluminación y en la calidad de la segmentación. . . . 317

6.3 Comparación del texto extraído por Tesseract a partir de cuatro versiones de la misma imagen: (a) imagen original; (b) CLAHE seguido de umbralización por Otsu; (c) normalización de fondo seguida de umbralización por Otsu; y (d) normalización de fondo en tonos de gris, sin umbralización. La comparación evidencia que la binarización externa no siempre favorece el reconocimiento, ya que Tesseract ya realiza su propia binarización adaptativa internamente. . 320
6.4 Flujo de reconocimiento y traducción automática. La imagen preprocesada por normalización de fondo, sin umbralización, se utiliza como entrada para Tesseract OCR, y el texto reconocido se traduce del inglés al español mediante la biblioteca *deep-translator*. 321
6.5 *Pipeline* de ingesta de documentos: rasterización adaptativa de páginas PDF a matrices discretas en formato PNG, mostrando la página dos. 324
6.6 Simulador interactivo de corrección de inclinación (*deskew*): mueva el *slider* para inclinar el documento y observe las tres etapas del *pipeline* — imagen inclinada, bordes Canny y resultado corregido. 327
6.7 *Pipeline* de rectificación axial: exhibición comparativa entre la entrada rotacionada original, el mapa de gradientes estructurales de Canny y el resultado final alineado con fondo normalizado en blanco. 328
6.8 Detección de los discos marcadores, extracción de contornos y rectificación por transformación de perspectiva. 330
6.9 Simulador interactivo de filtrado por circularidad: mueva el *slider* para ajustar el umbral C y observe qué componentes son aceptados (verde) o rechazados (rojo). 331
6.10 Pipeline de procesamiento del *QRCode*: umbralización, apertura morfológica, inversión y recorte final para decodificación. 334
6.11 Simulador interactivo del *pipeline* de aislamiento del QRCode: navegue por las etapas de filtrado, ajuste el elemento estructurante del cierre morfológico y vea la detección del contorno cuadrado sobre la imagen original de la plantilla. 335
6.12 Decodificación de código de barras lineal con *pyzbar*: imagen de entrada y datos extraídos. . . 337
6.13 Imagen de la hoja de respuestas en escala de grises cargada a partir del PDF rasterizado. . . 339
6.14 Área de respuestas extraída por *getAnswerArea*: región rectificada que contiene los cuadros de marcado. 340
6.15 Región del *QRCode* aislada por *segmentQRcode* dentro del área de respuestas rectificada. . . 341
6.16 Recorte inferior del área de respuestas, concentrando los cuadros de burbujas a ser segmentados.342
6.17 Respuestas leídas automáticamente por MCTest tras la segmentación y clasificación de todas las burbujas. 344
6.18 Detección de defecto por sustracción de imagen: producto de referencia, imagen con defecto simulado y máscara de anomalía detectada. 346
6.19 Simulador interactivo de inspección industrial por sustracción de imágenes: controle las distorsiones geométricas de desalineación (registro) y el umbral de detección para observar el impacto en los falsos positivos. 347
6.20 Detección de heterogeneidad de textura: mapa de varianza local y máscara de anomalía. . . . 349
6.21 Simulador EP06_01: IoU entre máscara de referência e máscara predita 356
6.22 Simulador EP06_02: Filtro de Marcadores por Circularidad 358
6.23 Simulador EP06_03: Clasificación de Marcaciones OMR 360
6.24 Simulador EP06_04: Estimador de Pendiente por Mediana Angular 362
6.25 Simulador EP06_05: Normalización de Fondo por División 364
6.26 Simulador EP06_06: Mapa de Variância Local para Detecção de Textura 366
6.27 Simulador EP06_07: *Pipeline* de Inspección — Registro por Traslación y Sustracción 368
6.28 Simulador EP06_08: Segmentación Geométrica + Verificación por Decodificación de Código QR372

7.1 Ejemplo motivacional: diferentes frutas formando agrupaciones distintas en un espacio de características. 378

7.2 Panorama del *pipeline* clásico de clasificación de imágenes: extracción de descriptores (LBP, HOG), formación del espacio de características y clasificación mediante k-NN. No se aplica a modelos de Aprendizaje Profundo (CNNs, YOLO), que aprenden *features* de extremo a extremo directamente de los píxeles. **Fuente:** elaborado con ayuda de Gemini Notebook ({GOOGLE}, 2025). . . . 379
7.3 Comparación visual de diferentes descriptores aplicados a la misma imagen. Cada descriptor revela aspectos distintos de la escena. . . . 381
7.4 Histograma de los códigos LBP de la imagen de los camarógrafos. Cada barra representa la frecuencia relativa de un código LBP, formando el vector de características utilizado para describir la textura. . . . 382
7.5 Primeros 100 componentes del vector de características HOG. . . . 384
7.6 Importancia de la normalización de las características para el clasificador k-NN. . . . 387
7.7 Mapa conceptual del proceso de clasificación de imágenes utilizando descriptores clásicos (LBP, HOG, píxeles brutos) y clasificador tradicional (k-NN). No se aplica a modelos de Aprendizaje Profundo (CNNs, YOLO), que aprenden *features end-to-end* directamente de los píxeles. . . . 388
7.8 Matrices de confusión obtenidas por el clasificador k-NN utilizando tres descriptores diferentes. Las filas representan la clase real (Manzana, Plátano y Naranja) y las columnas la clase predicha. Cuanto mayor sea la concentración de valores en la diagonal principal, mejor será el desempeño del descriptor. . . . 391
7.9 Comparación detallada de precisión global en escenario realista. Observe cómo los descriptores extraídos estructuralmente (LBP y HOG) superan el uso de intensidades puras de píxeles brutos. 392
7.10 Ilustración del proceso de generación de datos sintéticos con *make_classification*. A la izquierda, visualización de las tres clases en un espacio bidimensional formado por dos características informativas. A la derecha, importancia relativa de cada característica para la clasificación, evidenciando que solo 7 de las 10 características son efectivamente discriminativas, mientras que las demás son redundantes (2) o repetidas (1). . . . 395
7.11 Clasificación de una nueva muestra por el algoritmo k-NN. El punto de prueba (estrella) se clasifica a partir de los tres vecinos más cercanos, resaltados por círculos. . . . 398
7.12 Simulador interactivo de la frontera de decisión del k-NN: añada puntos de entrenamiento y ajuste el valor de k para observar el efecto sobre la región de decisión. . . . 399
7.13 Muestra de dígitos manuscritos de la base *load_digits*, utilizada como caso de estudio de clasificación. . . . 400
7.14 Matriz de confusión del clasificador k-NN entrenado con vectores de intensidad brutos (píxeles). 401
7.15 Comparación de precisión entre descriptores de píxeles brutos y HOG para el clasificador k-NN (k=3) en la base de dígitos. . . . 402
7.16 Precisión media por validación cruzada (5 particiones) en función del parámetro k, para la base de dígitos con vectores de intensidad brutos. . . . 404
7.17 Precisión en los conjuntos de entrenamiento y prueba para diferentes valores de k. Las regiones coloreadas representan, de forma conceptual, tendencias de comportamiento del clasificador: mayor riesgo de sobreajuste (rojo), compromiso entre sesgo y varianza (verde) y mayor riesgo de subajuste (azul). . . . 406
7.18 Muestras sintéticas de las tres clases de textura utilizadas en el experimento de clasificación, generadas con ruido gaussiano y variabilidad intra-clase. . . . 408
7.19 Precisión media obtenida mediante validación cruzada (5-fold) para los descriptores de Píxeles Brutos, LBP y HOG aplicados a la base de texturas sintéticas. . . . 410
7.20 Métricas de evaluación detalladas para el clasificador k-NN con descriptores LBP . . . 413
7.21 Simulador EP07_01: Clasificador k-NN Paso a Paso . . . 419
7.22 Simulador EP07_02: Efecto de la Normalización *Z-score* en la Distancia k-NN . . . 421
7.23 Simulador EP07_03: Precisión x Revocación . . . 423
7.24 Simulador EP07_04: Código LBP de una Vecindad 3×3 . . . 426
7.25 Simulador EP07_05: Histograma HOG de una Celda (mapeo de ángulos a bins) . . . 429
7.26 Simulador EP07_06: *Pipeline* k-NN Multi-Clase (votación, desempate y matriz de confusión) 432

7.27 Mosaicos de referencia (formato PGM ASCII) utilizados en el EP07_07. Caso 1: cuadrícula 2x2 con una muestra de cada clase. Caso 2: cuadrícula 3x3 con clases repetidas y mayor variabilidad. 437
7.28 Simulador EP07_07: Clasificación de un Mosaico de Texturas vía LBP + k-NN 437

8.1 Visión general de los principales conceptos abordados en este capítulo, incluyendo correspondencia de características, detección de objetos, segmentación de imágenes y sus relaciones con enfoques modernos basados en *Deep Learning*. **Fuente:** elaborado con ayuda de Gemini Notebook ({GOOGLE}, 2025). 440
8.2 Comparación del desempeño de diferentes implementaciones de la distancia de Hamming. . . 444
8.3 Simulador interactivo del descriptor ORB: explora la lógica de rotación y construcción del descriptor binario. Cambia la rotación para observar cómo el patrón de muestreo de pruebas binarias del BRIEF (líneas verdes y rojas) se orienta dinámicamente para garantizar la invariancia angular. 444
8.4 Imagen original generada por Gemini y versión rotacionada (20°), simulando un cambio de punto de vista. 445
8.5 Las 5 mejores correspondencias de características ORB entre la imagen original y la escena sintética, antes del filtrado por RANSAC. 447
8.6 Simulador interactivo del algoritmo RANSAC: ajuste el umbral de distancia y ejecute el algoritmo para observar la separación entre *inliers* y *outliers*. 448
8.7 Simulador interactivo de RANSAC aplicado al registro de imágenes: los puntos clave de dos imágenes se emparejan mediante un descriptor, algunas correspondencias son espurias (valores atípicos), y RANSAC estima la transformación de similitud que alinea la mayoría de ellas. . . 449
8.8 Correspondencias clasificadas como *inliers* (verde) por RANSAC al estimar la homografía entre las dos imágenes. 450
8.9 Registro de la imagen de la escena utilizando la homografía inversa estimada por RANSAC. . 451
8.10 Detección de caras y ojos con *Haar Cascade*. 454
8.11 Efecto de la Supresión de No-Máximos (NMS): múltiples detecciones redundantes (izquierda) se reducen a una detección por objeto (derecha). 456
8.12 Segmentación clásica (no basada en aprendizaje profundo): máscara semántica (moneda x fondo) mediante umbralización de Otsu, y segmentación de instancias mediante etiquetado de componentes conectados. 459
8.13 Separación de monedas que se tocan mediante transformada de distancia + *watershed*, sin erosión del contorno de las monedas. 461
8.14 *Bounding boxes* reconstruidas a partir del archivo de anotaciones YOLO exportado por *mm.saveMeasures*, superpuestas a la imagen original de las monedas. 465
8.15 Simulador EP08_01: Distancia de Hamming entre Dos Descriptores Binarios 473
8.16 Simulador EP08_02: RANSAC — Votación por Inliers entre Modelos Candidatos 475
8.17 Simulador EP08_03: Soma Rectangular en O(1) — Múltiples Situaciones de Borde 477
8.18 Simulador EP08_04: IoU y Supresión de No-Máximos 479
8.19 Simulador EP08_05: Etiquetado de Componentes Conectados — Conectividad 4 vs. 8 481
8.20 Simulador EP08_06: Extracción de *Bounding Boxes*, Centroides y Propiedades con *mm.measure* 483
8.21 Simulador EP08_07: Morfología con Conectividad Configurable y Medición 486
8.22 Simulador EP08_08: Filtrado de Mediana en Tonos de Gris y Medición de Objetos OpenCV 488
8.23 Simulador EP08_09: Gradiente de Iluminação, Limiar Adaptativo y Medición OpenCV . . . 489
8.24 Simulador EP08_10: Separación de Discos Tangentes mediante la Transformada de Distancia L2 y *Watershed* 491
8.25 Simulador EP08_11: Clasificación Geométrica con Controles Independientes de *Overlays BBox* (GT y DET) 494
8.26 Simulador EP08_12: Imagem utilizada no EP08_12. À esquerda, a imagem original. À direita, a imagem com as *bounding boxes* do arquivo de gabarito. 497
8.27 Simulador EP08_12: Precisión de Segmentación en Múltiples Objetos (IoU) 498

9.1 Visión general de los principales conceptos abordados en este capítulo. **Fuente:** elaborado con ayuda de *Gemini Notebook* ({GOOGLE}, 2025). 500
9.2 Simulador interactivo de convolución 2D clásica: elige entre las tres imágenes sintéticas de entrada 12×12 (casa, cara feliz o triste) y un filtro 3×3 (Sobel V, Sobel H, Nitidez o Identidad) y avanza paso a paso para observar cómo los productos internos locales del campo receptivo construyen el mapa de características célula a célula. 504
9.3 Simulador interactivo de la Neurona Artificial en contexto de CNN: alterne entre una neurona de capa convolucional (donde x_i son intensidades de píxel en un campo receptivo y w_i son pesos del *kernel*) y una neurona de capa totalmente conectada, ajustando entradas, pesos, sesgo y función de activación para visualizar el cálculo de z y de la salida y en tiempo real. . 505
9.4 Simulador interativo da Camada Convolucional: escolha entre três imagens de entrada 12×12 (casa, rosto feliz ou rosto triste) para observar como os mesmos *kernels* fixos reagem a diferentes bordas e formas. Navegue pelo campo receptivo com os botões ou reprodução automática, ajuste a função de ativação e alterne o zero-padding, acompanhando o mapa de características sendo revelado célula a célula, com o cálculo detalhado termo a termo e a fórmula da dimensão de saída em tempo real. 507
9.5 Simulador interactivo de la función de activación ReLU: arrastre el control para ver cómo los valores negativos se ponen a cero y los valores positivos se conservan, tanto en la curva como en un mapa de características real. 508
9.6 Simulador interactivo de *pooling*: elija entre *max-pooling* y *average-pooling* y avance paso a paso para observar la reducción de la resolución espacial del mapa de características. 509
9.7 Simulador interactivo de la Función de Pérdida (*Cross-Entropy*): seleccione la clase real de la imagen (Casa, Feliz o Triste) y ajuste las probabilidades estimadas por la Softmax para visualizar el cálculo de la penalización escalar y el gráfico del logaritmo negativo en tiempo real. 510
9.8 Simulador interactivo de *Backpropagation*: avance los pasos de la Regla de la Cadena para acompañar el flujo de la señal de error en el sentido inverso de la red, observando el cálculo de las derivadas parciales del gradiente en cada capa. 510
9.9 Simulador interactivo de los Algoritmos de Optimización: compare la trayectoria del SGD y del Adam sobre una superficie de pérdida no convexa con mapa de calor y curvas de nivel. La línea sólida muestra el optimizador seleccionado avanzando paso a paso; la línea discontinua muestra, para comparación instantánea, el camino completo que el otro optimizador recorrería desde el mismo punto inicial. Observe cómo el SGD queda atrapado en el Mínimo Local a la derecha, mientras que el Adam puede o no cruzar la barrera central dependiendo del impulso acumulado y de la tasa de aprendizaje. Haga clic en cualquier punto del mapa para restablecer el punto inicial de los pesos. 512
9.10 Arquitectura simplificada de una CNN para clasificación de imágenes, destacando el *forward pass* y las etapas de entrenamiento mediante la función de pérdida, la retropropagación y el algoritmo de optimización. 513
9.11 Simulador interactivo de la arquitectura de una CNN: elige una de las imágenes de entrada 12×12 (casa, rostro feliz o triste), haz clic en cada bloque del *pipeline* — Entrada, *Conv+ReLU*, *Pooling*, *Flatten*, *FC* y *Softmax* — y ejecuta pasos de entrenamiento reales (*forward pass* + retropropagación) para observar la pérdida y la precisión evolucionando, los *kernels* siendo ajustados y el *Softmax* empezando a apuntar a la clase correcta. 514
9.12 Simulador interactivo del cálculo del gradiente de un peso del *kernel* convolucional. 517
9.13 Muestras de dígitos del conjunto de entrenamiento tras conversión a *tensores* PyTorch y normalización. 520
9.14 Representación del flujo de transformaciones dimensionales de los *tensores* a lo largo de la arquitectura *CNNDigitos*. 520
9.15 Curvas de entrenamiento y evaluación de la CNN en la base de dígitos: evolución de la pérdida de entropía cruzada en el conjunto de entrenamiento y de la exactitud en el conjunto de prueba a lo largo de 50 épocas. 524
9.16 Flujo de activaciones de la CNN entrenada al procesar un ejemplo real del dígito 3, del *dataset load_digits*: dimensiones de los tensores capa a capa, de la entrada al *logit* de salida. 525

9.17 Grafo computacional de la *CNNDigitos* generado vía *torchviz*, mostrando las operaciones de *forward* y los nodos de gradiente (*backward*) asociados a cada parámetro entrenable. 526
9.18 Grafo detallado de la clase CNNDigitos generado via *torchviz*. 528
9.19 Comparación de precisión entre los clasificadores clásicos del Capítulo 7 (píxeles brutos y HOG con k-NN) y la CNN entrenada en este capítulo, en la misma base de dígitos. 529
9.20 Muestras de los conjuntos de entrenamiento tras preprocesamiento y adecuación dimensional a los tensores PyTorch: Dominio A (dígitos 0 a 4, tarea de origen) y Dominio B (dígitos 5 a 9 reindexados a 0 a 4, tarea de destino). 532
9.21 Comparação de acurácia no conjunto de teste do Domínio B (dígitos 5 a 9) sob restrição de dados (20 exemplos de treino): demonstração do impacto do congelamento rígido e da transferência negativa em redes de baixa capacidade. 536
9.22 Flujo de activaciones y transformaciones dimensionales de los tensores en el modelo de transferencia de aprendizaje al procesar una muestra de prueba del Dominio B (dígito 7, reindexado a clase 2). 537
9.23 538
9.24 540
9.25 541
9.26 Muestras sintéticas del *dataset* de diagnóstico foliar: hoja sana, hoja enferma (manchas oscuras) y síntoma desconocido (clorosis amarillenta) con ruido sal y pimienta. 544
9.27 546
9.28 547
9.29 550
9.30 Muestras iniciales del *dataset* sintético ruidoso de objetos geométricos con las cajas delimitadoras del formato YOLO superpuestas. 553
9.31 Comparación entre la imagen original con ruido tipo sal y pimienta y la imagen después de la aplicación del Filtro de Mediana (3x3). Las *bounding boxes* en el formato YOLO permanecen inalteradas. 555
9.32 558
9.33 560
9.34 Muestras del conjunto de datos sintético de nódulos: imagen en escala de grises bajo ruido y su respectiva máscara binaria de referencia mostradas. 562
9.35 Línea de base clásica de segmentación: suavizado, umbralización de Otsu y apertura morfológica mostradas. 563
9.36 Evolución del entrenamiento de la U-Net compacta: reducción de la pérdida y elevación del índice IoU medio en el conjunto de validación a lo largo de 35 épocas. 567
9.37 Comparativo cuantitativo (IoU promedio) y cualitativo entre el enfoque clásico y la U-Net entrenada en tres muestras de validación. 568
9.38 571
9.39 572
9.40 573
9.41 Realidad aumentada basada en marcador ArUco: marcador insertado en escena real y rotado, detectado y sustituido por imagen virtual mediante homografía. 575
9.42 Medición de dimensiones físicas reales utilizando una tarjeta de referencia de escala conocida (8,56 cm) renderizada. El rectángulo gris representa la tarjeta de referencia y el rectángulo azul representa el objeto objetivo. Las dimensiones estimadas del objeto se muestran en centímetros. 578
9.43 Simulador EP09_01: Convolución 2D Manual (correlación cruzada + sesgo + ReLU, con *padding* y *stride* ajustables) 586
9.44 Simulador EP09_02: Pooling Manual (máximo vs. média, con ventana k y stride s ajustables) 589
9.45 Simulador EP09_03: Conteo de Parámetros — Convolución vs. Capa Totalmente Conectada 594
9.46 Simulador EP09_04: IoU y Supresión de No-Máximos (NMS) 597
9.47 Simulador EP09_05: Evaluación de Segmentación — IoU y Dice Píxel a Píxel 600
9.48 Simulador EP09_06: Pipeline Integrado — Detección a la Medición del Mundo Real 603

A.1 Questão do tabuleiro de xadrez gerada pelo MCTest, ilustrando o enunciado parametrizado com o valor de `height` sorteado para a variação e o exemplo de entrada/saída do primeiro caso de teste. 610
A.2 Questão real gerada pelo MCTest para o Simulado 4 — tópico "Operadores Morfológicos". . . 612

Lista de Tablas

1 Diferentes formatos de publicación generados automáticamente a partir del mismo código fuente. 4
2 Número de páginas del PDF y tiempo de renderizado por combinación (paralelismo real medio de ~5 procesos, pico de 8). La combinación base en Python/Portugués solo renderiza las salidas ya registradas en los *notebooks*-fuente; las demás reejecutan todo el código. 4

1.1 Conexión entre PDI, VC y otras áreas de la ciencia. 5
1.2 Principales tipos de imagen digital y sus respectivos conjuntos de valores posibles para cada píxel. 9
1.3 Principales bibliotecas y módulos utilizados a lo largo de este libro para computación matricial, PDI, VC, visualización y validación de los ejercicios. 9

2.1 Convenciones de escala y orden de canales en las principales bibliotecas de PDI. 52
2.2 Comparativo entre estrategias de compactación y eficiencia de procesamiento. 58
2.3 Comparativa de métricas de distancia aplicadas a la malla de píxeles. 60
2.4 Principales formatos de almacenamiento de imágenes digitales y sus aplicaciones en PDI. . . 60
2.5 Métodos de interpolación disponibles en `mm.resize` y sus respectivos números de vecinos utilizados en el cálculo. 66

3.1 Algoritmo de ecualización de histograma. 104
3.2 Interpretación típica de los coeficientes del *kernel*. 112
3.3 Etapas del *Unsharp Masking*. 126
3.4 Media vs. mediana con un píxel corrompido. La mediana ignora el valor atípico; la media se desplaza ~40 niveles. 129

4.1 Técnicas de preprocesamiento evaluadas para mejorar la separación entre monedas y fondo antes de la aplicación del método de Otsu. 162
4.2 Propiedades de apertura y cierre. 173
4.3 Secuencias del filtro secuencial alternado mm.asf. 174
4.4 Comparación entre los operadores *clohole* y *edgeoff*. 182
4.5 Taxonomía simplificada de los principales enfoques de segmentación y refinamiento estudiados en este capítulo. 189
4.6 *Pipeline* completo del *watershed* basado en marcadores para imágenes reales. 199
4.7 *Pipeline* simplificado utilizado en el ejemplo didáctico de la Figura 4.25. 202
4.8 Comparación entre los enfoques basados en componentes conexos y en contornos. 204

5.1 Correspondencia entre las regiones del espectro de magnitud tras la aplicación del FFT Shift. 246
5.2 Comparación de la complejidad de la convolución directa, separable y vía Transformada Rápida de Fourier (FFT). 252
5.3 Subbandas producidas por la Transformada Wavelet Discreta 2D (DWT), indicando los filtros aplicados en cada dirección y el contenido predominante de cada componente. 265
5.4 Comparación entre familias de *wavelets*, destacando la longitud del soporte, el número de momentos nulos, la simetría y las aplicaciones típicas. 266
5.5 Comparación entre la Transformada Discreta de Fourier (DFT) y la Transformada *Wavelet* Discreta (DWT), destacando sus principales características y aplicaciones. 274
5.6 Categorías de redundancia en imágenes digitales y sus respectivos mecanismos de explotación. 275
5.7 Etapas del *pipeline* de compresión JPEG y sus respectivos fundamentos de diseño. 279
5.8 Comparación estructural entre los principales formatos de imagen rasterizados. 283

5.9 Síntesis de las etapas del *pipeline* de compresión JPEG y sus respectivos impactos. 290

6.1 Imágenes de skimage.data utilizadas en los experimentos de este capítulo. 314
6.2 Selección de imágenes públicas representativas disponibles en el módulo *skimage.data*, organizadas por área de aplicación. 314

7.1 Conjunto de descriptores cromáticos, texturales y geométricos utilizados para representar imágenes de frutas. 392
7.2 Ejemplo simplificado de clasificación de frutas utilizando dos características normalizadas. . . 395

8.1 Comparación entre los principales paradigmas de segmentación visual. 457
8.2 Arquitecturas representativas de Aprendizaje Profundo para la detección y segmentación de imágenes. 466

9.1 Comparativa entre las principales tareas de VC según la granularidad de la información producida. 500
9.2 Comparación entre los tres escenarios de transferencia de aprendizaje presentados en esta sección. 548

C.1 Expressões de filtro de URL utilizadas na configuração do SEB. 615

PDI y VC — Python

Bienvenido al libro PDI y Visión por Computador — versión Python / Español.

Organización

- **Parte I — Fundamentos de PDI** — Representación, histogramas, filtrado, morfología
- **Parte II — Visión por Computador** — Segmentación, descriptores, detección, aprendizaje profundo

Prefacio

Este libro constituye una **obra en permanente actualización** en las áreas de **Procesamiento Digital de Imágenes (PDI)** y **Visión Computacional (VC)**, concebida como material didáctico interactivo para cursos de grado y posgrado en Computación, Ingeniería y áreas afines.

> **!** Destacado
>
> La obra se fundamenta en la metodología descrita en Zampirolli et al. (2025) — extensión de Zampirolli et al. (2024), trabajo premiado en la línea **Recursos y Ambientes Educativos** de la **EduComp 2024**. El contenido integra la biblioteca `morph.py`, desarrollada por el autor.

Este no es un libro estático. Su contenido evoluciona continuamente a medida que los ejemplos se mejoran, se incorporan nuevas secciones y los enfoques pedagógicos se refinan con base en la experiencia de uso y en la retroalimentación de estudiantes y docentes. De esta manera, la obra se trata como un *proyecto en constante evolución*, buscando acompañar tanto los avances tecnológicos como las mejores prácticas de enseñanza en PDI y VC.

Contexto de la primera edición

El contenido de esta edición fue elaborado entre mayo y agosto de 2026, durante la primera oferta de la asignatura de Procesamiento Digital de Imágenes basada en este material didáctico. La asignatura se impartió en dos grupos de grado —mayoritariamente compuestos por estudiantes de Ciencias de la Computación, en el período matutino— y en un grupo de posgrado en Ciencias de la Computación, todas en la UFABC. Esta primera edición representa el resultado de esa oferta inicial, consolidando el contenido desarrollado, probado y continuamente perfeccionado a lo largo del período lectivo.

La metodología de enseñanza adoptada siguió una secuencia estructurada en cada clase. Inicialmente, se proyectaba un video corto, con una duración media de 7 minutos, generado en **Gemini Notebook**, alternando entre la presentación de la parte conceptual del capítulo y la resolución de los Ejercicios de Programación (EP). En este segundo caso, casi todos los EP se resolvían durante la propia clase. A continuación, se presentaba un conjunto de aproximadamente 12 *diapositivas*, también generadas en **Gemini Notebook**, teniendo como fuentes el PDF completo del libro y el archivo `morph.py`. Estas *diapositivas* se producían a partir de un *prompt* específico para la generación del contenido teórico y práctico de cada capítulo.

Tras esta etapa inicial, quedaban aproximadamente 100 minutos de clase para la exploración de los dos *cuadernos* Colab del capítulo, uno teórico y otro práctico, con énfasis en los simuladores interactivos y en la ejecución de los bloques de código, permitiendo a los estudiantes modificar parámetros y observar sus efectos. En las clases realizadas en laboratorio, los estudiantes transferían las respuestas de los EP desarrolladas en Colab directamente a las actividades VPL de Moodle, en las cuales se sometían a la evaluación automática. Este proceso de publicación y utilización de los EP se presenta al final de este prefacio.

La evaluación continua incluyó simulacros quincenales, aplicados con **SEB (*Safe Exam Browser*)** en Moodle, que contenían EP similares a los de las listas de ejercicios. Cada simulacro otorgaba un bono del 5% sobre la nota final. Las dos pruebas de la asignatura también utilizaron SEB, pero estuvieron compuestas por preguntas parametrizadas generadas en **MCTest**, cuya descripción combina LaTeX y parámetros en el formato `[[code:variable]]`, definidos en fragmentos de Python delimitados por `[[def: ... ]]` en la propia pregunta. Este proceso permitió generar 110 variaciones de examen, sorteadas individualmente entre los estudiantes.

El Apéndice A detalla la creación de estas preguntas en MCTest y su exportación a Moodle; el Apéndice B describe la configuración de las actividades VPL, incluido el proceso de publicación de los EP; el Apéndice C presenta la configuración de SEB para los simulacros y los exámenes; el Apéndice D describe la generación de los videos y *diapositivas* utilizados en las clases con **Gemini Notebook**; y el Apéndice E detalla el uso de **Inteligencia Artificial (IA)** en la generación de *retroalimentación* socrática para actividades formativas y evaluativas, complementando la corrección automática realizada por VPL.

Cómo se produce este libro

El contenido de este libro se desarrolla en **Quarto** y se almacena en la carpeta `all` del repositorio **github.com/fzampirolli/pdi-vc**. A partir de un único código fuente, el libro se genera automáticamente y se publica en diferentes formatos, presentados en Tabla 1.

Aunque la edición en PDF registra el estado de la obra al término de la primera oferta de la disciplina en 2026, el desarrollo del libro continúa de forma permanente. Con cada actualización del repositorio de GitHub, se generan automáticamente nuevas versiones de las páginas HTML, del PDF y de los notebooks, poniendo las mejoras inmediatamente a disposición de los lectores.

Tabla 1: Diferentes formatos de publicación generados automáticamente a partir del mismo código fuente.

Formato	Descripción
HTML	Versión para *web*, con simuladores interactivos y navegación entre capítulos: fzampirolli.github.io/pdi-vc/
PDF	Versión adecuada para impresión o lectura *offline*: livro.pt.py.pdf
Notebooks (`.ipynb`)	Compatibles con **Jupyter** y **Google Colab**, que permiten ejecutar, modificar y experimentar con los ejemplos de código y los Ejercicios de Programación (EPs). Un filtro personalizado mantiene las referencias cruzadas, la numeración de figuras y tablas, además de formatear automáticamente las citas según el estándar ABNT.

La obra se publica en **diez combinaciones** — cada ruta de código (**Python** y **C++**) en cinco idiomas (portugués, inglés, francés, español e italiano). Con la caché de traducción ya poblada, una regeneración completa (traducción, reejecución de los *notebooks*, renderizado de HTML y PDF y publicación) tarda unos **72 minutos**, con un paralelismo real medio de ~5 procesos (pico de 8); la **primera** generación de un idioma nuevo, con la caché vacía, es mucho más lenta — unos **80 minutos** por combinación, como se observó en las primeras generaciones de español e italiano. Tabla 2 resume cada combinación (ya con la caché poblada): número de páginas del PDF y tiempo de renderizado. Este tiempo total real es mucho menor que la suma de los tiempos de cada combinación por separado (más de **5 horas** si se ejecutaran una por una): en una CPU con muchos núcleos, varias combinaciones se procesan al mismo tiempo, por lo que el tiempo total no es la simple suma de los tiempos individuales.

Tabla 2: Número de páginas del PDF y tiempo de renderizado por combinación (paralelismo real medio de ~5 procesos, pico de 8). La combinación base en Python/Portugués solo renderiza las salidas ya registradas en los *notebooks*-fuente; las demás reejecutan todo el código.

Combinación	Ruta	Idioma	Páginas	Tiempo de render.
`py.pt`	Python	Portugués (base)	654	~7 min
`py.en`	Python	Inglés	644	~31 min
`py.fr`	Python	Francés	666	~31 min
`py.es`	Python	Español	666	~31 min
`py.it`	Python	Italiano	658	~31 min
`cpp.pt`	C++	Portugués	434	~26 min
`cpp.en`	C++	Inglés	426	~34 min

Combinación	Ruta	Idioma	Páginas	Tiempo de render.
`cpp.fr`	C++	Francés	434	~38 min
`cpp.es`	C++	Español	430	~37 min
`cpp.it`	C++	Italiano	428	~35 min

Estado de validación. Solo la combinación `py.pt` (Python, portugués) es la fuente curada: es donde el autor escribe, revisa y valida todo el contenido. Las otras nueve combinaciones — las rutas en **C++** y las traducciones al inglés, francés, español e italiano — se **generan automáticamente**, mediante traducción con un modelo de lenguaje y transpilación de código, y todavía **requieren una revisión detallada**. El punto más sensible es el **texto incrustado en algunas figuras**: donde aún no se ha producido la versión traducida, la imagen aparece con texto en portugués (cada figura traducida sigue el mismo sufijo de idioma que los demás archivos generados, por ejemplo `imagen_en.png`).

Las páginas HTML, el PDF y los *notebooks* disponibles en el proyecto reflejan siempre el estado más reciente del desarrollo. Cuando se publica una **edición oficial**, esta queda identificada mediante una *etiqueta* (tag) en el repositorio de GitHub, lo que permite reproducir exactamente el contenido correspondiente a esa edición. Así, el lector puede seguir tanto la evolución continua del proyecto como recuperar cualquier edición oficial publicada anteriormente.

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

Cada capítulo dispone de dos botones **Ejecutar en Colab** (Figura 1), que dirigen a entornos complementarios:

1. **Parte Teórica**, ubicada al inicio de cada capítulo, que contiene los conceptos presentados y ejemplos de código ejecutables;
2. **Parte Práctica**, en la sección **Parte Práctica con Ejercicios de Programación** (EPs), dedicada a los ejercicios y a sus simuladores interactivos.

Figura 1: Botón clicable **Ejecutar en Colab**, disponible al inicio de las partes **Teórica** y **Práctica** de cada capítulo, que permite la ejecución interactiva de los ejemplos y ejercicios. En la versión PDF, existe un *enlace* HTML directo entre la imagen del simulador y su leyenda.

La generación de los *notebooks* está totalmente automatizada mediante el *script* `gerar_notebooks_alunos.py`, que adapta el contenido a entornos interactivos, permitiendo su ejecución sin necesidad de instalar Quarto.

Uso de herramientas de Inteligencia Artificial

La concepción, el proyecto pedagógico, la estructura conceptual y el contenido fundamental de este libro son de **autoría exclusiva del autor**.

En el proceso de edición y apoyo al desarrollo, se utilizaron, en sus versiones gratuitas, herramientas de **Inteligencia Artificial (IA)**, como **ChatGPT**, **Claude**, **DeepSeek** y **Gemini**, empleadas estrictamente como recursos auxiliares. Su uso se limitó al apoyo en la revisión y mejora del estilo textual, a la optimización de la sintaxis de códigos y a la generación asistida de ilustraciones conceptuales y ejemplos.

Todas las respuestas y contenidos sugeridos por estas herramientas fueron sometidos a **análisis crítico, verificación, validación técnica, adaptación e integración por parte del autor**, quien asume la responsabilidad por el texto, los códigos y los recursos pedagógicos presentados. Las herramientas de IA **no se consideran autoras ni coautoras de la obra**, ni responsables de las decisiones intelectuales, técnicas o pedagógicas que fundamentan su contenido.

Se destaca, además, que los **simuladores** y recursos interactivos presentes en el libro —disponibles en las versiones HTML y Jupyter Notebook (IPYNB)— fueron diseñados e implementados como parte del enfoque pedagógico de la obra, con el objetivo de proporcionar experimentación directa de los conceptos presentados y favorecer la autonomía de aprendizaje del lector.

A diferencia del uso editorial descrito anteriormente, el *pipeline* de generación multilingüe (ver "Cómo se produce este libro") emplea un modelo de lenguaje como **componente de ingeniería del sistema**: la traducción automática del contenido entre portugués y otros idiomas, con caché incremental que evita retraducir fragmentos sin cambios. Esta etapa utiliza la API de pago de **DeepSeek** (modelo `deepseek-v4-flash`); el libro completo ya ha sido traducido del portugués al **inglés**, al **francés**, al **español** y al **italiano**, y los **Capítulos 1 a 5** cuentan además con una ruta en **C++** que compila y ejecuta de verdad — a un costo acumulado del orden de unos pocos dólares, gracias a la caché incremental.

La biblioteca `morph.py`

La biblioteca `morph.py` (Zampirolli et al., 2025) acompaña toda la obra como una herramienta de apoyo a la enseñanza. Su objetivo no es sustituir bibliotecas consolidadas, como **OpenCV**, **scikit-image**, **scikit-learn**, **NumPy** o **Matplotlib**, ni competir con ellas en rendimiento o alcance. En cambio, busca **hacer transparentes los algoritmos fundamentales de Procesamiento Digital de Imágenes y Visión Computacional (PDI-VC)**, permitiendo que el estudiante comprenda su implementación, modifique el código y desarrolle nuevas funcionalidades.

Siempre que sea posible, la biblioteca ofrece dos versiones para un mismo algoritmo: una **implementación didáctica**, escrita paso a paso para reflejar directamente las definiciones matemáticas presentadas en el libro, y otra **implementación optimizada**, basada en bibliotecas especializadas para aplicaciones prácticas.

> **Herencia Técnica**
>
> La estructura de `morph.py` tiene como base herramientas de Morfología Matemática desarrolladas en Brasil, como **MMachLib** (Lotufo et al., 1997) y **MMach** (Barrera et al., 1998), además de la **mmorph**, utilizada en la obra de Dougherty; Lotufo (2003).. Estas bibliotecas sirvieron de referencia para la enseñanza del área durante décadas.

Esta filosofía puede observarse, por ejemplo, en los operadores morfológicos de dilatación:

- `mm.dil`: implementación optimizada, indicada para aplicaciones prácticas;
- `mm.dil0`: implementación didáctica de la dilatación para **elementos estructurantes planares**, siguiendo directamente la definición clásica de la Morfología Matemática;
- `mm.dil1`: implementación didáctica de la dilatación para **funciones estructurantes (*kernels* no planares)**, siguiendo rigurosamente su formulación matemática.

La biblioteca también ofrece funciones para lectura, visualización, conversión de colores, filtrado y morfología matemática a través de una interfaz simple:

```
from morph import mm

img = mm.gray(mm.read("lena.jpg"))  # lectura y conversión a niveles de gris
grad = mm.gradm(img)                # gradiente morfológico
mm.show(grad)                       # visualización
```

A lo largo del libro, se introducen nuevos métodos en la biblioteca para simplificar la realización de los Ejercicios de Programación (EPs). Entre ellos están `mm.readTrain()` y `mm.readTest()`, que automatizan la lectura de las bases de entrenamiento y prueba utilizadas en los capítulos dedicados al reconocimiento de patrones.

Además, todos los proyectos del **Gemini Notebook** incluyen el archivo `morph.py`, permitiendo consultar y discutir directamente la implementación de los algoritmos presentados en el libro. De esta forma, el estudiante

puede utilizar el propio Gemini Notebook como un asistente de estudios, haciendo preguntas como:

> Explique cómo se implementó el algoritmo `mm.knn0` de `morph.py`, mostrando el código y comentando cada etapa de forma didáctica. Además, explique las diferencias entre `mm.knn0()` y `mm.knn()`.

! Implementaciones didácticas e implementaciones optimizadas

En varios capítulos, un mismo algoritmo se presenta en dos versiones. Las funciones con sufijo `0`, `1`, etc. (por ejemplo, `mm.knn0()`, `mm.dil0()` y `mm.dil1()`) son implementaciones didácticas, desarrolladas para facilitar la comprensión de los algoritmos. En cambio, las funciones sin sufijo (como `mm.knn()` y `mm.dil()`) utilizan implementaciones optimizadas, frecuentemente basadas en bibliotecas especializadas, como OpenCV, scikit-image y scikit-learn, siendo más adecuadas para aplicaciones prácticas.
En las actividades evaluadas por el **VPL de Moodle**, bibliotecas como **scikit-learn** y, en algunos casos, **scikit-image**, pueden no ser utilizadas debido a las limitaciones de memoria del entorno de ejecución. Por este motivo, los Ejercicios de Programación (EPs) emplean preferentemente las implementaciones didácticas de `morph.py`, que producen los mismos resultados, aunque con menor rendimiento computacional. Esta restricción no existe cuando los programas se ejecutan localmente, por ejemplo, en **VS Code**, **Jupyter Notebook** o **Google Colab**.

El código fuente de la biblioteca está disponible en:

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

Ejercicios de Programación (EPs) y Validación Automática

Cada unidad del libro incluye **Ejercicios de Programación (EPs)** prácticos y de complejidad creciente, diseñados para consolidar los conceptos presentados a lo largo del capítulo. Cada EP está acompañado por un **simulador interactivo**, disponible en las versiones HTML e IPYNB, que permite al estudiante manipular parámetros, visualizar el comportamiento de los algoritmos y desarrollar una comprensión intuitiva del problema antes de iniciar su implementación. De esta forma, el proceso de aprendizaje combina experimentación, programación y validación automática.

La validación de las soluciones se realiza localmente mediante la clase `TestSuite` (`testsuite.py`), que compara la salida del programa con los archivos de casos de prueba (`.cases`):

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

El sistema soporta múltiples lenguajes (Python, Java, C++, C, JavaScript y R) y puede integrarse directamente en Moodle. Para docentes interesados en el paso a paso completo de publicación de los EPs como actividades VPL, consulte el **Guía del Profesor**, al final de este prefacio, y el Apéndice B.

Código abierto

El proyecto se rige por principios de código abierto. El repositorio público reúne el texto, los códigos, las imágenes y los *scripts*: **github.com/fzampirolli/pdi-vc**

Cada versión del libro se archiva permanentemente en **Zenodo** con un *DOI citable*. Para citar este material en trabajos académicos:

> ZAMPIROLLI, Francisco de Assis. **PDI+VC — Procesamiento Digital de Imágenes y Visión Computacional**. UFABC, 2026. DOI: 10.5281/zenodo.20784605

Cabe registrar, al respecto, que el contenido expuesto en esta obra refleja la mirada crítica, la propuesta pedagógica y la experiencia docente de su autor. Así, los análisis, enfoques y opiniones expresados a lo largo de este libro representan únicamente el entendimiento de su creador, no constituyendo ni reflejando un posicionamiento oficial o institucional de la Universidad Federal del ABC (UFABC).

Antes de comenzar: *Notebooks* en Python

El concepto de ***Literate Programming*** (Programación Literaria), propuesto por Donald Knuth (Knuth, 1984), fundamenta la estructura de este material. La lógica invierte el paradigma tradicional: el programa se escribe para la lectura humana, asemejándose a un ensayo, mientras que el código se extrae por separado para la ejecución computacional.

El contenido está estructurado en *notebooks* — documentos que intercalan **células de texto** (en Markdown) y **células de código** (en Python).

- **Ejecución:** las células de código se identifican con `[ ]`. La ejecución puede realizarse con `Shift + Enter` o mediante el botón ▷ de la interfaz.
- **Entornos:** los *notebooks* pueden ejecutarse localmente, mediante Jupyter o Visual Studio Code (VS Code), así como en entornos de nube, como Google Colab.

> **Nota sobre el formato**
>
> En entornos interactivos, el código puede modificarse y ejecutarse. En las versiones estáticas (HTML o PDF), los bloques de código tienen finalidad de lectura y referencia, sin perjuicio de la integridad de las explicaciones.

Guía del Profesor: Publicación de EPs en Moodle

*Esta sección está destinada a docentes que deseen integrar los Ejercicios de Programación (EPs) a las actividades **VPL (Virtual Programming Lab)** de Moodle, complementando el Apéndice B.*

Integración con Moodle (VPL)

Los EPs de los *notebooks* pueden convertirse en actividades **VPL** en Moodle. El *script* `ep_tools.py` (ejecutado mediante `make build`) automatiza la exportación de los EPs del final de cada capítulo en dos etapas:

1. **Extracción** (`make eps`): genera un HTML interactivo independiente por EP, con enunciado, ejemplos y simulador, en `gen/book/eps/<versao>/`. Vea la lista completa en: https://fzampirolli.github.io/pdi-vc/eps/py.pt/index.html
2. **Conversión** (`make moodle`): transforma el HTML en un fragmento autocontenido, sin dependencia de CSS externo, listo para pegarse en el editor de Moodle, en `gen/book/eps/<versao>_moodle/`. Vea un ejemplo en: https://fzampirolli.github.io/pdi-vc/eps/py.pt_moodle/EP01_01.html

Adicionalmente, todos los simuladores interactivos desarrollados a lo largo del libro pueden accederse directamente en https://fzampirolli.github.io/pdi-vc/simuladores/py.pt/.

> **Otras rutas e idiomas**
>
> Los enlaces anteriores usan `py.pt` como ejemplo. Para acceder a los EPs o simuladores de otra combinación lenguaje/idioma, basta con cambiar ese segmento en la URL — por ejemplo, `cpp.it` para la ruta C++ en italiano: `https://fzampirolli.github.io/pdi-vc/eps/cpp.it/index.html`. La estructura es la misma para todas las combinaciones disponibles, en `/eps/<versión>/`, `/eps/<versión>_moodle/` y `/simuladores/<versión>/`.

Al publicar con `make publish`, estas **dos versiones** de cada EP quedan disponibles en los *enlaces* indicados. El profesor puede combinar las dos estrategias: pegar la versión Moodle en el editor VPL (con simulador funcional) e incluir en el enunciado un *enlace* a la versión completa, donde las fórmulas se renderizan correctamente mediante MathJax.

Revisión obligatoria antes de publicar en Moodle

Aunque automatizada, la conversión exige revisión del profesor debido a las limitaciones del editor TinyMCE/HTML Purifier de Moodle:

- **Fórmulas matemáticas** — TinyMCE descarta barras invertidas (\), corrompiendo ecuaciones LaTeX. Por ello, la versión Moodle convierte fórmulas simples a HTML puro (entidades como `≥`, `×`). Las fórmulas complejas (`\begin{cases}`, matrices, integrales) pueden salir incompletas — revise y, si es necesario, reescríbalas en texto o indique la versión completa mediante *enlace*.
- **Figuras externas** — Las imágenes complementarias deben insertarse manualmente en la actividad VPL.
- **Simuladores** — Funcionan perfectamente siempre que utilicen JavaScript estándar con estilos *inline* y manipulación directa del DOM (`getElementById`). **Atención:** si incluye fórmulas en LaTeX que contengan el carácter \ en Moodle, los simuladores pueden dejar de funcionar.

Cómo Publicar en Moodle

1. Acceda a la versión Moodle del EP deseado:

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

2. Copie el código fuente completo de la página (`Ctrl+U` → `Ctrl+A` → `Ctrl+C`).
3. En Moodle, cree la actividad VPL, acceda al editor HTML, pegue el contenido (`Ctrl+V`) y guarde.
4. Importe los archivos `.cases` de la carpeta `all/capXX/casos/` en las configuraciones de pruebas del VPL.

Reutilización de los Casos de Prueba

Los archivos `.cases` son compatibles con el VPL, permitiendo usar el mismo conjunto de pruebas tanto en el desarrollo local como en la corrección automatizada en Moodle.

Parte I

Parte I — Fundamentos de PDI

Capítulo 1

Fundamentos y Primeros Pasos

Executar Colab Abrir GitHub

Este capítulo inaugura la **Parte 1** del libro, dedicada a los fundamentos del Procesamiento Digital de Imágenes (PDI). Se presentan la representación matemática de imágenes digitales y los principales métodos para su manipulación.

Se utiliza el lenguaje Python y la biblioteca `morph.py` (ZAMPIROLLI, 2025).

1.1 Objetivos

Al final de este capítulo, usted será capaz de:

- Comprender la naturaleza física y matemática de la imagen digital $f(x, y)$.
- Identificar las bandas del espectro electromagnético relevantes para PDI.
- Realizar operaciones básicas: lectura, visualización y guardado de imágenes.
- Acceder y modificar intensidades de píxeles individualmente.
- Aplicar umbralización manual.
- Configurar el entorno de desarrollo en Python.
- Manipular estructuras de arreglos (NumPy) sin caer en trampas de memoria.

1.2 Antes de comenzar: *Notebooks* interactivos

Este material fue construido bajo el concepto de ***Literate Programming*** (Programación Literaria), ideado por Donald Knuth en la década de 1980 (KNUTH, 1984). Knuth —también creador del sistema **TeX** para tipografía digital— propuso que los programas se escribieran como una narrativa lógica para los seres humanos, intercalando código y documentación.

Para ejecutar una celda, presione Shift + Enter o haga clic en el botón ▷.

> **Nota sobre el formato**
>
> En las versiones renderizadas (PDF o HTML), el código se presenta en bloques estáticos con fines de lectura y referencia. La ejecución interactiva requiere el acceso a través de Google Colab (disponible en la parte superior de la página) o en un entorno local mediante VSCode o Jupyter Notebook.

1.3 Fundamentos

El estudio de sistemas basados en imágenes comprende un ecosistema de disciplinas integradas que transforman datos visuales brutos en conocimiento estructurado. Mientras algunas áreas se centran en la generación

de representaciones, otras se dedican al tratamiento y análisis de estos datos para respaldar aplicaciones tecnológicas complejas.

El diagrama presentado en el Figura 1.1 establece la distinción y complementariedad entre el Procesamiento Digital de Imágenes (PDI) y la Visión por Computador (VC). El PDI, destacado en **verde**, se centra en la transformación de imagen a imagen, con el objetivo de mejorar la calidad o realizar un preprocesamiento, como la eliminación de ruidos y el realce de contraste.

En contraste, la VC, señalada en **azul**, se centra en la interpretación del contenido visual para extraer modelos o información, como el reconocimiento de objetos y gestos. La región de intersección ilustra la sinergia entre las áreas, donde el PDI prepara los datos visuales para la interpretación por parte de la VC. El mapa también demuestra las interconexiones de ambas disciplinas con áreas como Robótica, Computación Gráfica, Inteligencia Artificial (IA) y Neurociencia.

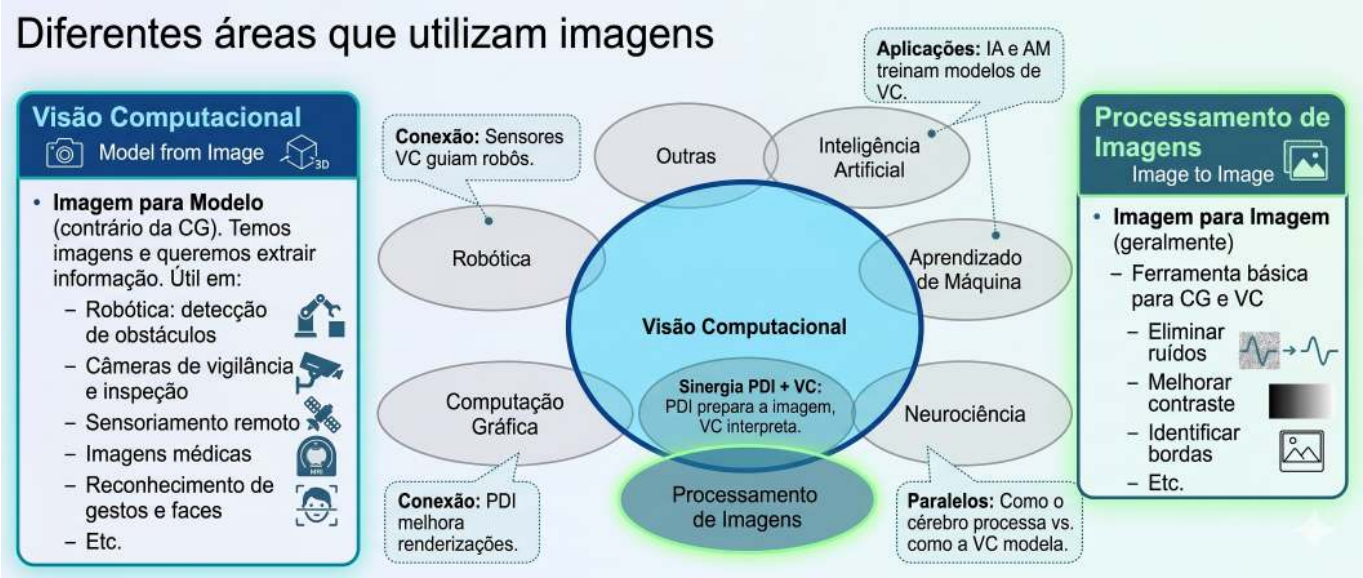


Figura 1.1: Diagrama relacional que detalla las distinciones fundamentales, sinergias e interconexiones entre PDI y VC en el contexto de sistemas basados en imágenes.

1.3.1 Visión por Computador

- **Enfoque:** *Imagen* → *Modelo* (camino inverso de la Computación Gráfica).
- **Objetivo:** Extraer información de alto nivel a partir de imágenes o videos.
- **Aplicaciones típicas:**
  - Robótica: detección de obstáculos, localización y navegación autónoma.
  - Vigilancia e inspección: reconocimiento de eventos, lectura de placas, control de calidad.
  - Teledetección: análisis de imágenes satelitales, cartografía ambiental.
  - Imágenes médicas: detección de tumores, segmentación de órganos, ayuda al diagnóstico.
  - Interacción humano-computador: reconocimiento de gestos, expresiones faciales, seguimiento ocular.
- **Relación con otras áreas:** utiliza técnicas de Aprendizaje Automático e IA para clasificar e interpretar escenas; sirve como "ojos" de la Robótica.

1.3.2 Procesamiento Digital de Imágenes (PDI)

- **Enfoque:** *Imagen* → *Imagen* (generalmente: transformación de una imagen en otra).
- **Objetivo:** Mejorar la calidad visual o extraer características de bajo nivel.

- **Aplicaciones comunes:**
  - Eliminación de ruido (filtros de media, mediana, gaussiano).
  - Mejora del contraste (ecualización de histograma, ajuste gamma).
  - Detección de bordes (Sobel, Canny, Laplaciano).
  - Segmentación (umbralización, crecimiento de regiones, *watershed*).
  - Transformaciones geométricas (redimensionamiento, rotación, corrección de perspectiva).
- **Relación con otras áreas (ver Tabla 1.1):**
  - Es la **base** de la mayoría de los sistemas de VC (preprocesamiento).
  - La Computación Gráfica aplica frecuentemente PDI para posprocesamiento (ej.: suavizado, realce).
  - Las técnicas de IA pueden optimizar parámetros de procesamiento (ej.: aprendizaje de filtros).

Tabla 1.1: Conexión entre PDI, VC y otras áreas de la ciencia.

Área	Relación con PDI y VC
Inteligencia Artificial	Proporciona modelos (redes neuronales, SVM) que interpretan salidas de la VC.
Robótica	Consume datos de VC para tomar decisiones (navegación, manipulación).
Aprendizaje Automático	Utiliza descriptores extraídos por PDI/VC para entrenar clasificadores.
Computación Gráfica	Camino inverso: *modelo* $\rightarrow$ *imagen*; muchas veces aplica PDI para renderizado realista.
Neurociencia	Inspira modelos de PDI (ej.: filtros similares a células ganglionares de la retina).

1.4 Etapas del PDI

Las etapas del PDI se presentan en la Figura 1.2, que pueden comprenderse como una cadena de transformaciones que reduce la redundancia de los datos en busca de significado:

- **Bajo Nivel:** Actúa directamente sobre los píxeles de la imagen ruidosa para realizar mejoras y filtrados, generando como salida una imagen limpia o realzada.
- **Nivel Medio:** Recibe la imagen tratada y realiza la segmentación y descripción, transformando la matriz de píxeles en atributos estructurados (forma, tamaño y textura).
- **Alto Nivel:** Utiliza la tabla de atributos para alimentar procesos de lógica e inteligencia artificial, resultando en la decisión o reconocimiento final (como el diagnóstico médico).

A Figura 1.3 detalla la secuencia completa del PDI, desde la captura hasta la interpretación. El flujo comienza en la Adquisición de la Imagen (1) y prosigue con el Mejoramiento (2) y la Restauración (3). A continuación, el contenido se aísla mediante la Segmentación (4) y se refina con la Morfología (5). La transición crucial ocurre en la Representación y Descripción (6), donde los objetos visuales se convierten en datos matemáticos (área, perímetro, etc.), lo que permite el Reconocimiento (7). Los procesos auxiliares incluyen el Procesamiento de Imagen en Color y la Compresión, que contribuyen a la eficiencia del almacenamiento y del análisis.

1.5 Formación de la Imagen y el Espectro

El proceso de formación de una imagen se fundamenta en la interacción entre la materia y la energía radiante. Esencialmente, una imagen se concibe cuando un **sensor registra la radiación** resultante de la interacción con un objeto físico. En el contexto de la visión humana y de la fotografía convencional, este fenómeno depende de una fuente de luz que ilumine la escena; las características de los objetos se codifican entonces a través de las **variaciones de intensidad y color** de la luz que alcanza el sensor, tal como se ilustra en la Figura 1.4.

La **luz visible** ocupa solo una pequeña franja del espectro electromagnético — entre **380 nm (violeta)** y **750 nm (rojo)** — según se ilustra en Figura 1.5.. Los sensores digitales convencionales operan en esa

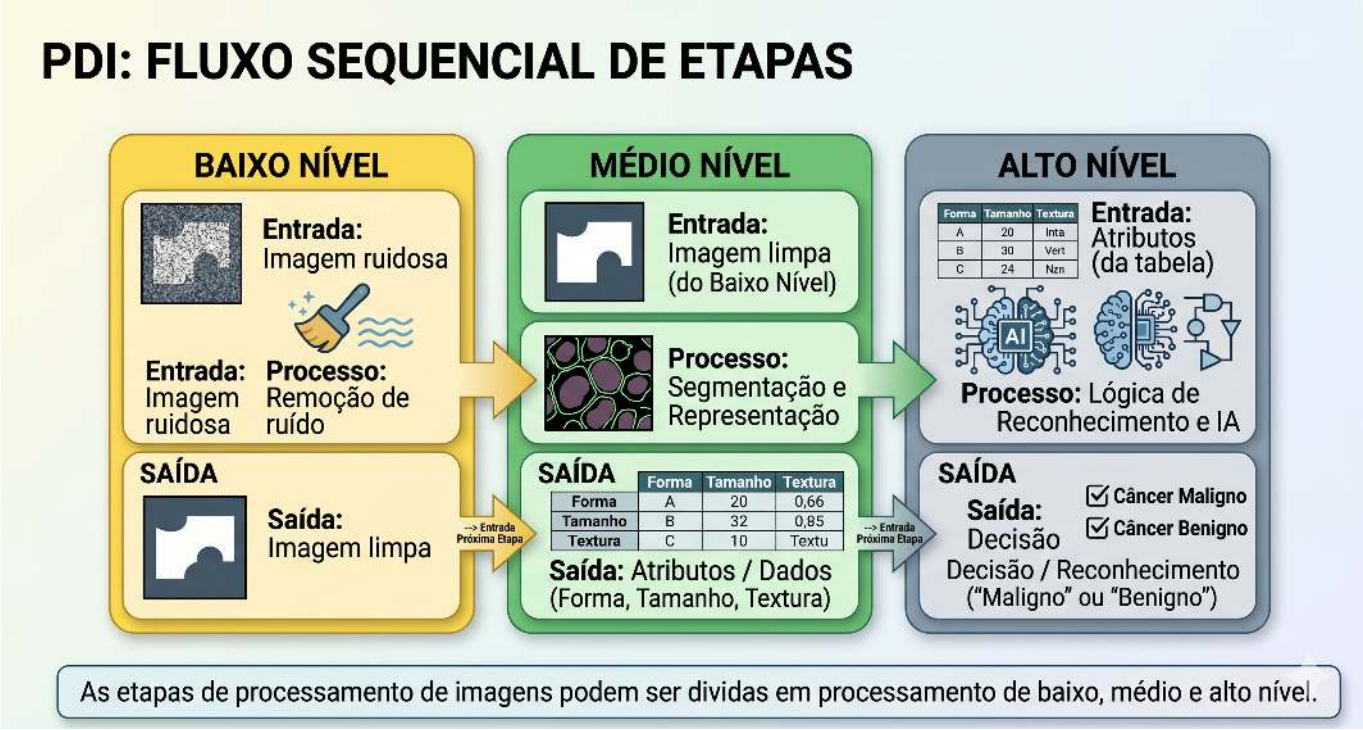


Figura 1.2: Representación del flujo secuencial de procesamiento: la salida de cada nivel se convierte en la entrada del nivel subsiguiente.

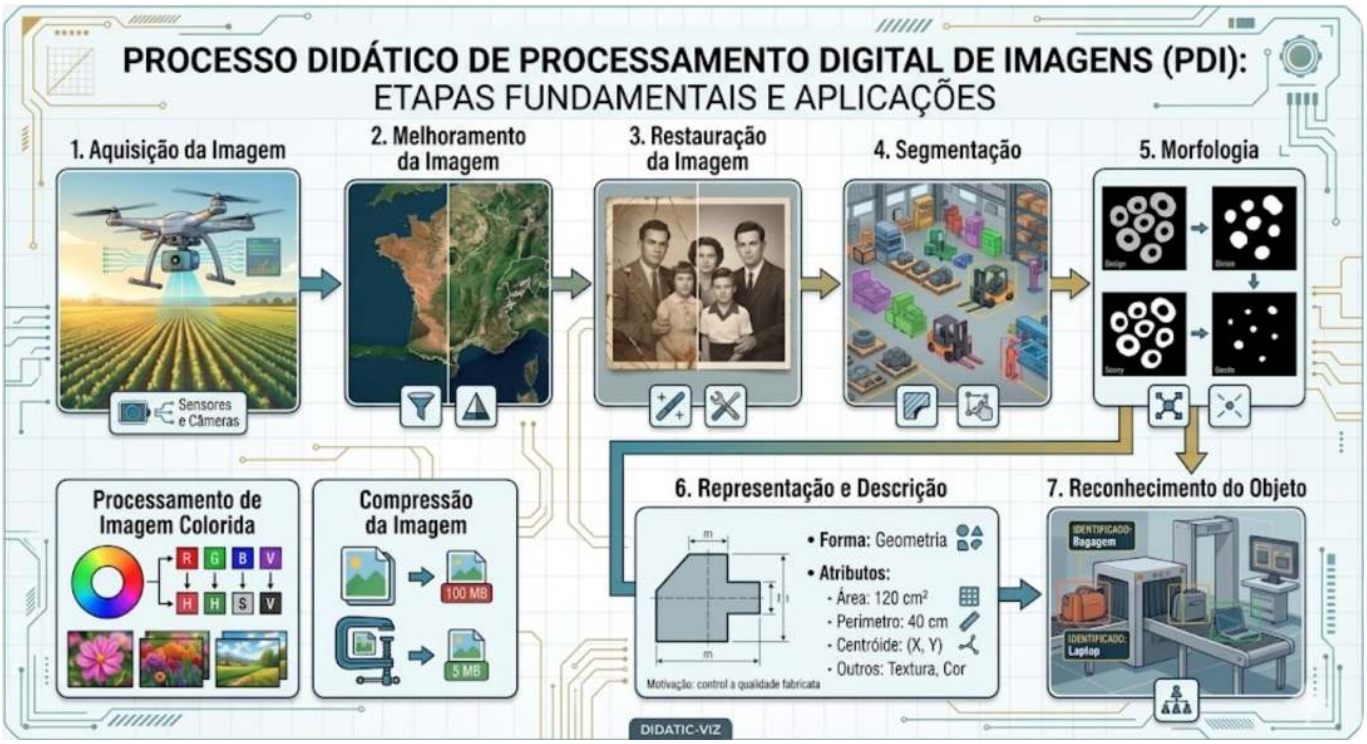


Figura 1.3: Flujo detallado del PDI: desde la adquisición sensorial hasta la extracción de atributos y el reconocimiento automatizado, incluyendo el procesamiento en color y la compresión.

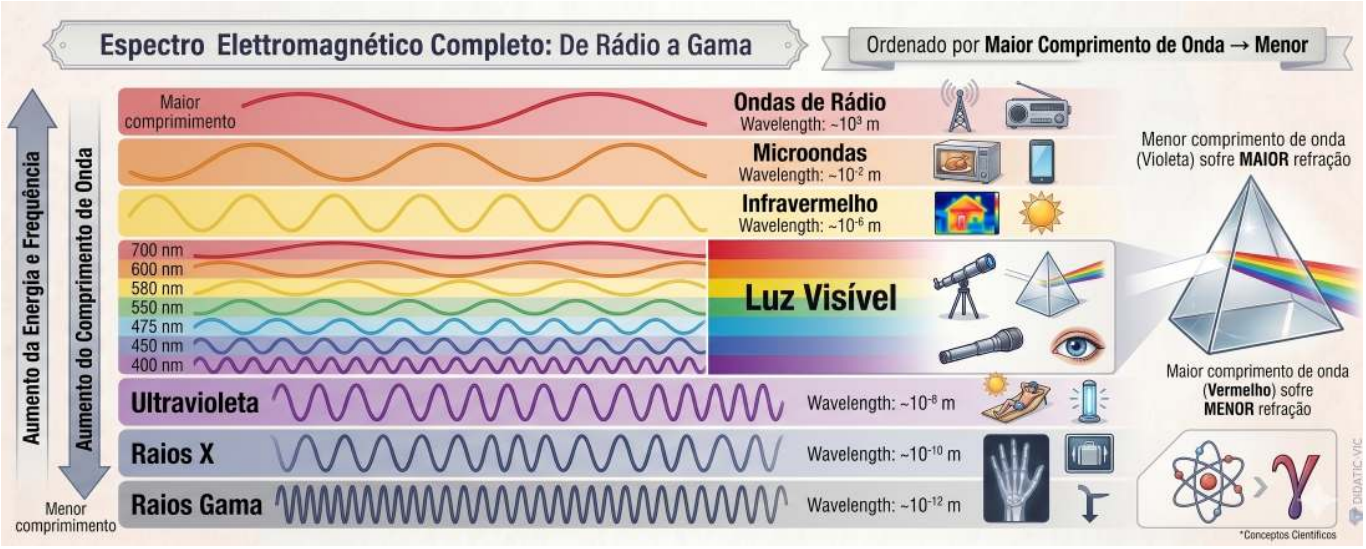


Figura 1.4: Representación del espectro visible y su posición en relación con las demás radiaciones electromagnéticas, destacando la variación de las longitudes de onda de 380 nm a 750 nm.

misma ventana, pero los equipos especializados pueden captar radiaciones invisibles al ojo humano, como el infrarrojo y los rayos X. En PDI, la imagen formada depende directamente de la sensibilidad espectral del sensor utilizado.

A partir de ese proceso físico de adquisición, se vuelve posible modelar matemáticamente la imagen digital como una función bidimensional discreta, en la cual cada punto de la escena está representado por muestras numéricas de intensidad luminosa, formalizando así los conceptos de píxel y de imagen digital presentados en la siguiente sección.

1.6 ¿Qué es una Imagen Digital?

Una **imagen digital** está formada por una cuadrícula de **píxeles** (*Picture Elements*), donde cada píxel es la unidad elemental más pequeña de la imagen.

> ¿Qué es un Píxel?
>
> Un **píxel** es la unidad direccionable más pequeña que compone una imagen digital. Cada píxel ocupa una posición única en la cuadrícula y almacena uno o más valores numéricos que representan su intensidad o color.

Representación Matemática

A diferencia de una función continua, el dominio de una imagen digital es un plano rectangular finito $\mathbb{E} \subset \mathbb{Z}^2$, que representa la cuadrícula de muestreo. Este dominio está indexado por coordenadas enteras:

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

Donde:

- L: representa el ancho de la imagen (número de columnas).
- H: representa la altura de la imagen (número de filas).

La imagen digital es una función que asocia cada par de coordenadas (x, y) a uno o más valores que describen la apariencia del píxel.

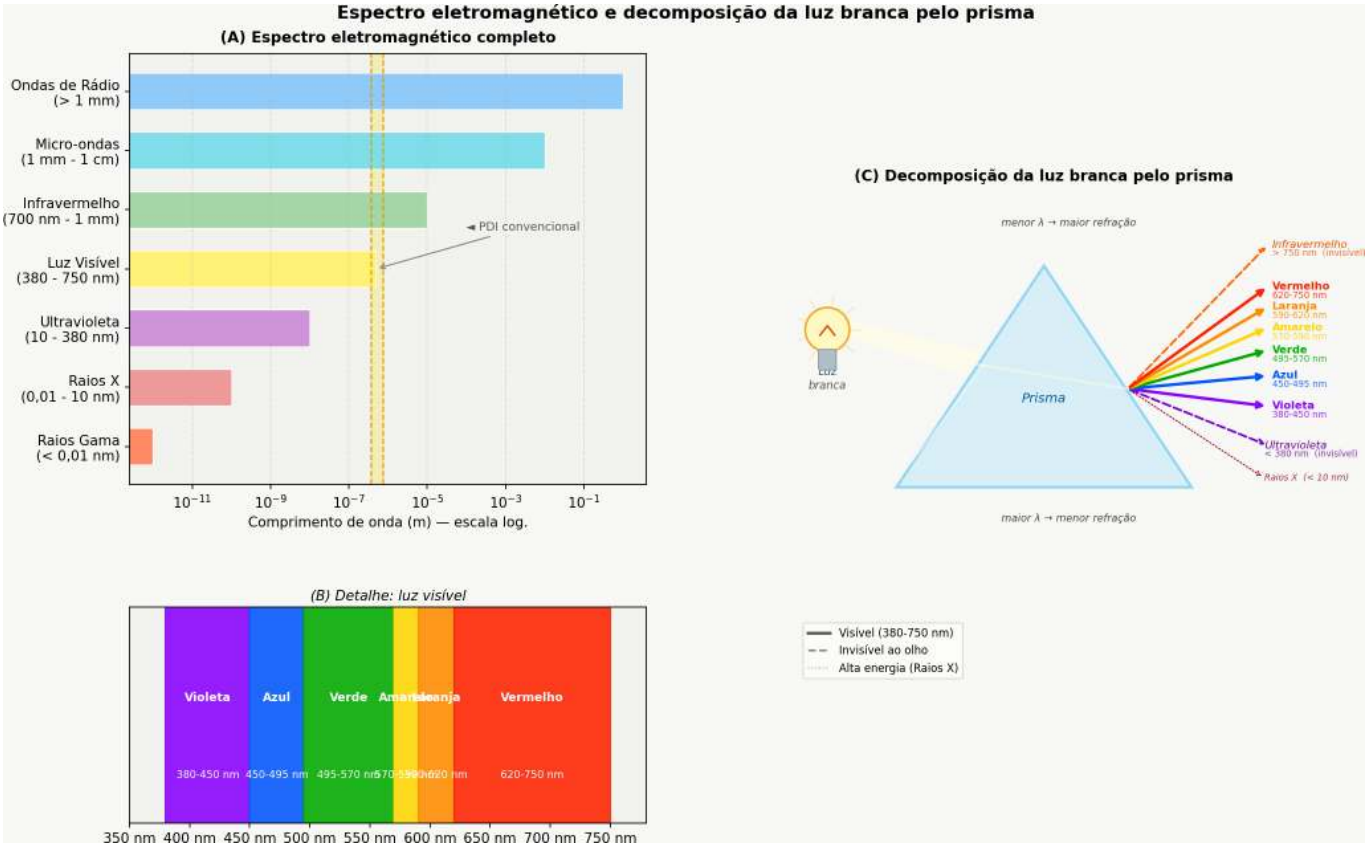


Figura 1.5: (A) Espectro electromagnético completo en escala logarítmica, con énfasis en el rango visible. (B) Detalle de la luz visible (380-750 nm) y sus colores. (C) Descomposición de la luz blanca por el prisma: la menor longitud de onda sufre mayor refracción, separando UV, visible e infrarrojo.

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

El conjunto $\mathcal{V}$ define los valores posibles para el píxel (codominio), variando según el tipo de imagen, como se demuestra en la Tabla 1.2.

Tabla 1.2: Principales tipos de imagen digital y sus respectivos conjuntos de valores posibles para cada píxel.

Tipo de imagen	$\mathcal{V}$ (valores del píxel)	Representación
Binaria	$\{0, 1\}$ o $\{0, 255\}$	■□
Escala de grises	$\{0, 1, \ldots, 255\}$	`[ ][=][#]`■
Color (RGB)	$\{0, \ldots, 255\}^3$ (triplas ordenadas de valores)	■■■

Ejemplo práctico: Una imagen a color en el modelo RGB puede representarse matemáticamente mediante una función que asocia tres valores de intensidad a cada píxel. Computacionalmente, esta representación corresponde a tres matrices superpuestas — los canales rojo (*Red*), verde (*Green*) y azul (*Blue*) — en las cuales cada elemento almacena la intensidad luminosa del respectivo canal en una posición determinada de la imagen.

Para viabilizar los experimentos prácticos de PDI y VC, este libro utiliza el ecosistema científico de Python, integrando bibliotecas orientadas a la manipulación matricial, al procesamiento de imágenes y a la visualización de resultados, presentadas a continuación.

1.7 Configuración del Entorno

Este material utiliza el ecosistema científico de Python, con énfasis en **NumPy** (computación matricial), **OpenCV** (visión computacional) y la biblioteca **morph.py** (ZAMPIROLLI, 2025), desarrollada con fines didácticos y utilizada a lo largo de este libro, según se presenta en la Tabla 1.3.

El proyecto está disponible actualmente en **Python** y **Portugués**. Su organización, sin embargo, permite su expansión a otros lenguajes de programación e idiomas.

Los **Ejercicios de Programación (EPs)** presentados al final de los capítulos pueden ser validados mediante el módulo `testsuite.py`, que permite ejecutar casos de prueba en diferentes lenguajes, incluidos Python, C, C++, Java, JavaScript y R. De esta forma, los mismos casos de prueba pueden utilizarse para verificar las soluciones desarrolladas en los *notebooks* y en **Moodle/VPL**.

Tabla 1.3: Principales bibliotecas y módulos utilizados a lo largo de este libro para computación matricial, PDI, VC, visualización y validación de los ejercicios.

Biblioteca	Función principal
`numpy`	Representación matricial y operaciones numéricas
`opencv-python`	Lectura, escritura y operaciones de visión computacional
`matplotlib`	Visualización de imágenes y gráficos
`morph.py`	Abstracción didáctica de operaciones de PDI
`testsuite.py`	Ejecución y validación automática de los EPs

1.8 Versiones de `morph.py`

La biblioteca `morph.py` tiene dos versiones públicas:

- **Versión 1.0** — versión original, publicada junto con el artículo presentado en EduComp 2024: repositorio de la versión 1.0

- **Versión 1.1** — versión utilizada en las actividades de este libro: repositorio de la versión 1.1

La versión 1.1 fue adaptada para atender también las restricciones de memoria del **Moodle/VPL** (Laboratorio Virtual de Programación). En la versión 1.0, algunas bibliotecas, como `matplotlib`, `requests` y `skimage`, se cargaban en el momento del `import morph`. En entornos con recursos restringidos, esa carga podía exceder el límite de memoria y producir errores como `Jail: out of memory, 128MiB`.

En la versión 1.1, esos *imports* fueron reemplazados por **carga *diferida* (*lazy*)** : cada biblioteca se importa únicamente mediante el método que efectivamente la utiliza. Así, el comando `import morph` mantiene la carga inicial reducida, utilizando principalmente `numpy` y `cv2`, mientras que los recursos adicionales se cargan bajo demanda.

La configuración de los *notebooks* se centraliza en el archivo `config.py`, disponible en el mismo repositorio. Ese archivo verifica las dependencias, procura garantizar el uso de **OpenCV 5.0** y proporciona `morph.py` y, cuando es necesario, `testsuite.py`. Cuando el entorno no cuenta con la versión adecuada de OpenCV, el `config.py` puede reinstalar la versión requerida y, si es necesario, solicitar el reinicio del entorno de ejecución.

De esta manera, los *notebooks* utilizan una única rutina de configuración, reduciendo la repetición de código y facilitando la reproducción del entorno computacional utilizado en este libro.

```
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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

1.9 Fundamentos de Matrices — atención a la copia de referencias

Como una imagen digital puede representarse mediante una matriz, es importante comprender cómo crear y manipular matrices correctamente. En Python, existe una trampa común al utilizar el operador `*` con listas:

> ⚠ Atención a la copia de referencias
>
> Al ejecutar `m = [[0]*2]*3`, no se crean tres filas independientes. En lugar de ello, se crean tres referencias a la **misma lista**. En consecuencia, la modificación de un elemento en una de las filas también modificará el elemento correspondiente en las demás filas.

Para visualizar este comportamiento, se puede ejecutar el código en Python Tutor y compararlo con la forma correcta de crear una matriz utilizando listas: `m = [[0]*2 for _ in range(3)]`.

En la práctica, para el procesamiento digital de imágenes (PDI), se recomienda utilizar **NumPy**, que proporciona estructuras de datos multidimensionales eficientes y operaciones vectorizadas adecuadas para la manipulación de imágenes. El siguiente código presenta diferentes formas de creación de imágenes sintéticas, cuyos resultados se muestran en la Figura 1.6..

```
import numpy as np

# Creando una imagen negra (ceros) de 4, 6 píxeles
img_preta = np.zeros((4, 6), dtype='uint8')
img_preta[0,0] = 255  # píxel blanco en la esquina superior izquierda
```

```
# Creando una imagen blanca (255) de 4, 6 píxeles
img_branca = np.ones((4, 6), dtype='uint8') * 255
img_branca[3,5] = 0  # píxel negro en la esquina inferior derecha

# Creando una imagen aleatoria para pruebas (ruido)
img_random = mm.randomImage(4, 6, maxValue=255)

print("Matriz aleatoria generada:")
print(mm.drawImg(img_random))

mm.show(
    [img_preta, img_branca, img_random],
    titles=[
        "Predominantemente negra\n(con 1 píxel blanco en (0,0))",
        "Predominantemente blanca\n(con 1 píxel negro en (3,5))",
        "Imagen aleatoria\n(simulación de ruido)"
    ],
    cols=3,
    figsize=(9, 3),
    axis=True
)
```

```
Matriz aleatoria generada:
 30  23 210  24 213 170
 43 147 113  63  67 239
203  50 144 228  28  28
146  14  15  76   2  50
```

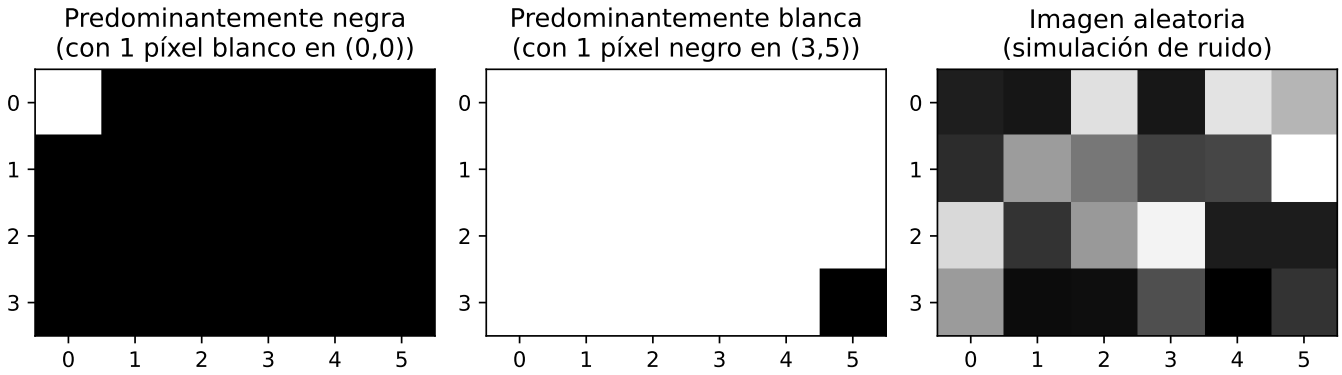


Figura 1.6: Ejemplos de imágenes sintéticas representadas matricialmente.

1.10 Leyendo y Mostrando Imágenes

En bibliotecas como NumPy, OpenCV y scikit-image, las imágenes digitales se representan frecuentemente de forma computacional como *arrays* de NumPy. De esta manera, las operaciones matriciales y los conceptos de álgebra lineal pueden aplicarse directamente al PDI.

Una de las operaciones fundamentales en PDI es la lectura de imágenes.

En la biblioteca `morph.py`, la función `mm.read()` permite cargar imágenes tanto de archivos locales como de URLs, como se ilustra en la Figura 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"Dimensiones (H, W, Canales): {img.shape}")
print(f"Tipo de dato: {img.dtype}")

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

```
Dimensiones (H, W, Canales): (1365, 2048, 3)
Tipo de dato: uint8
```

Figura 1.7: Mandril (Mandrillus sphinx) en ambiente natural. Crédito: Julien Renoult (CC BY 4.0).

Alternativa: *Descarga* de la Imagen para Almacenamiento Local

En entornos donde la lectura directa de URLs no esté disponible —debido a restricciones de red, políticas de *firewall* o ausencia de conectividad—, una alternativa consiste en realizar previamente la *descarga* de la imagen al sistema de archivos local. Tras el almacenamiento, la imagen puede cargarse normalmente mediante la función `mm.read()`. En el siguiente ejemplo, el archivo se guarda localmente con el nombre `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="Ejemplo: Captura RGB (archivo local)")
```

Explicación

- `wget -O mandrill.png <URL>` realiza la *descarga* de la imagen y la almacena localmente con el nombre especificado;
- `mm.read('mandrill.png')` efectúa la lectura del archivo directamente desde el sistema de archivos, sin necesidad de solicitudes HTTP adicionales;
- esta estrategia reduce la dependencia de conectividad durante la ejecución de los experimentos y evita *descargas* repetidas de la misma imagen.

Nota

El prefijo `!` se utiliza en entornos basados en *notebooks*, como Jupyter Notebook, JupyterLab y Google Colab, para ejecutar comandos del sistema operativo directamente en celdas de código. En terminales convencionales, el comando debe utilizarse sin el prefijo `!`.
Si la utilidad `wget` no está instalada, se puede usar alternativamente:

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

1.11 Conversión de Tipos y Umbralización

Como vimos, una imagen a color en el espacio RGB se representa mediante la función:

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

Es decir, para cada píxel (x, y), tenemos tres valores (R, G, B) que definen su color.

Conversión a Escala de Grises

Para convertir una imagen RGB a escala de grises (*grayscale*), es necesario combinar los tres canales en un único valor de intensidad g, que representa el brillo percibido. Como el ojo humano no es igualmente sensible al rojo, verde y azul, se utiliza una **media ponderada**. El estándar ITU-R BT.601 ({ITU-R}, 2011) define los siguientes pesos:

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

Tras el cálculo, el valor g se redondea al entero más cercano y se ajusta al intervalo $[0, 255]$. El resultado es una nueva imagen, ahora en escala de grises, representada por:

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

Umbralización (*Thresholding*)

A partir de la imagen en escala de grises $f_{\text{gris}}(x, y)$, una operación fundamental es la **umbralización**, que produce una imagen **binaria** (solo blanco y negro). Para ello, se elige un valor de corte T (generalmente en el intervalo $[0, 255]$) y se define:

$$f_{\text{bin}}(x, y) = \begin{cases} 255 & \text{si } f_{\text{gris}}(x, y) > T \\ 0 & \text{en caso contrario} \end{cases} \tag{1.4}$$

Ejemplo: Con $T = 128$, los píxeles con intensidad superior a 128 se vuelven blancos (255); los demás se vuelven negros (0).

La umbralización se usa ampliamente para **segmentar objetos del fondo**, extraer bordes o crear máscaras binarias para el procesamiento posterior.

> **Nota:** El valor 255 representa el blanco máximo en imágenes de 8 bits, mientras que 0 representa el negro absoluto.

Ejemplo práctico de conversión y umbralización

La Figura 1.8 ilustra los principales pasos para transformar una imagen a color en escala de grises y, posteriormente, convertirla en una imagen binaria mediante umbralización. El siguiente código implementa estas etapas:

```
# 1. Convertir a Escala de Grises
img_gray = mm.gray(img)

# 2. Aplicar umbral (Píxeles > 128 se convierten en 255, otros 0)
limiar = 128
img_binaria = mm.threshold(img_gray, limiar)

# Uso de la nueva función
mm.show(
    [img, img_gray, img_binaria],
    titles=["Original", "Escala de Grises", f"Binaria (T={limiar})"],
    cols=3
)
```

Figura 1.8: Procesamiento básico de imágenes: (a) imagen original, (b) imagen en escala de grises, (c) imagen binarizada por umbral (T=128).

1.12 Umbralización por el método de Otsu

Como se presentó en la Ecuación 1.4, la umbralización convierte una imagen en tonos de grises a una binaria usando un valor de corte T. Hasta ahora fijamos $T = 128$ manualmente.

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

Sin embargo, la elección manual de T no siempre es trivial. La biblioteca `mm` ofrece una alternativa automática: cuando el parámetro `limiar` **no se proporciona**, la función `mm.threshold(img_gray)` calcula el valor de T mediante el **método de Otsu** (OTSU, 1979). Este método, que se detallará en capítulos futuros, maximiza la varianza entre clases del histograma (frecuencia de cada tono de gris), separando automáticamente los píxeles de objeto y fondo.

El código siguiente compara la umbralización manual ($T = 128$) con la automática (Otsu), mostrando también el valor de T calculado:

```
import cv2
# Umbralización con T fijo (manual)
T_fixo = 128
img_bin_fixo = mm.threshold(img_gray, T_fixo)

# Umbralización por el método de Otsu (T automático)
T_otsu, img_bin_otsu = cv2.threshold(img_gray, 0, 255,
                                     cv2.THRESH_BINARY + cv2.THRESH_OTSU)
print(f'Umbral calculado por Otsu: T = {T_otsu}')
# o simplemente:
# img_bin_otsu = mm.threshold(img_gray)

# Exhibición lado a lado
mm.show(
    [img_gray, img_bin_fixo, img_bin_otsu],
    titles=["Escala de Grises", f"Binaria (T={T_fixo})", f"Binaria (Otsu, T={T_otsu})"],
    cols=3
)
```

```
Umbral calculado por Otsu: T = 95.0
```

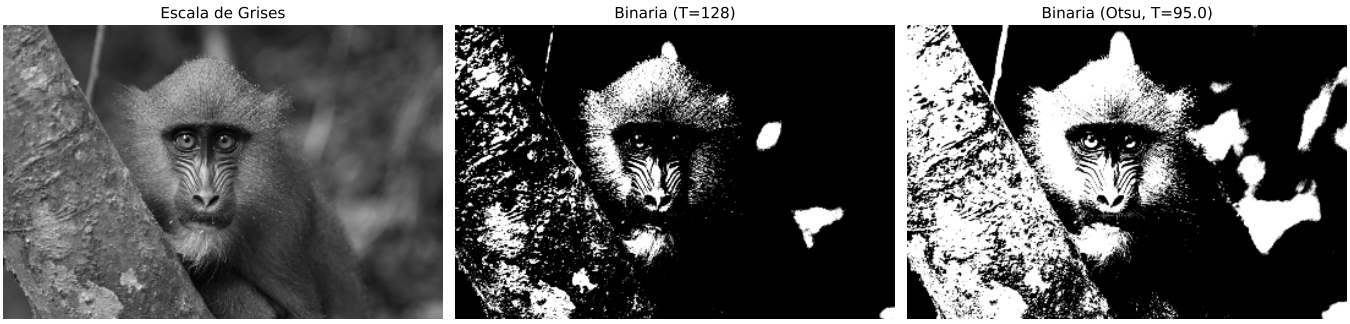


Figura 1.9: Comparación entre umbralización manual (T=128) y automática (Otsu) sobre la imagen en escala de grises.

La Figura 1.9 muestra que el umbral obtenido por Otsu se adapta automáticamente a la imagen, resultando en una binarización más eficiente que un valor fijo, especialmente cuando las intensidades del objeto y del fondo están bien separadas en el histograma. Esta técnica es ampliamente utilizada en sistemas de VC para la binarización de documentos, detección de objetos y preprocesamiento de imágenes.

Simplicidad de la biblioteca `morph.py`

Mientras que OpenCV exige la llamada completa:

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

la biblioteca `mm` abstrae toda esa complejidad: basta con llamar a `mm.threshold(img_gray)`. El umbral de Otsu se calcula automáticamente y la imagen binaria se devuelve directamente. Este enfoque permite concentrarse en el concepto, no en los detalles de implementación.

1.13 Acceso a Píxeles

En Python con NumPy, una imagen está estructurada como un **array multidimensional**. El acceso a un píxel específico utiliza la convención matricial **fila** (eje Y) y **columna** (eje X): `img[fila, columna]`.

El código de la Figura 1.10 demuestra cómo extraer estos valores en imágenes de color (RGB) y en escala de grises, además de aislar la vecindad inmediata del punto de interés.

```
import matplotlib.pyplot as plt

# 1. Definición de las coordenadas del píxel objetivo
r, c = 600, 800

# 2. Acceso directo a los valores de los píxeles
pixel_cinza = img_gray[r, c]
pixel_rgb   = img[r, c]

print(f"  VALORES EN EL PÍXEL OBJETIVO ({r}, {c}):")
print(f"    • Tonos de gris (Escalar) : {pixel_cinza}")
print(f"    • Colorido (Vector RGB)    : R={pixel_rgb[0]}, G={pixel_rgb[1]}, B={pixel_rgb[2]}")
print("-" * 50)

# 3. Exhibición didáctica del vecindario 3x3 alrededor del píxel (r, c)
# Recorta desde la fila r-1 hasta r+1, y desde la columna c-1 hasta c+1
vizinhanca_gray = img_gray[r-1:r+2, c-1:c+2]

print(f"  MATRIZ DE VECINDARIO 3x3 EN TONOS DE GRIS:")
print(f"    (El píxel objetivo central está resaltado entre corchetes)\n")

# Impresión formateada para resaltar el píxel central
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}]") # Resalta el centro (100, 100)
        else:
            linhas_str.append(f" {valor:3d} ")
    print("    " + " ".join(linhas_str))

# 4. Demostración visual usando el mm.show (opcional, pero excelente para el libro)
# Muestra la imagen completa con un punto demarcando la región
plt.figure(figsize=(5, 5))
plt.imshow(img)
plt.plot(c, r, 'ro', markersize=8, label=f'Pixel ({r},{c})') # Traza (x, y) -> (c, r)
plt.title(f"Localização do Pixel ({r}, {c}) na Imagem")
plt.legend()
plt.axis('on') # Mantiene los ejes para que el estudiante vea las coordenadas 100, 100
plt.show()
```

```
  VALORES EN EL PÍXEL OBJETIVO (600, 800):
    • Tonos de gris (Escalar) : 81
    • Colorido (Vector RGB)    : R=92, G=78, B=67
--------------------------------------------------
  MATRIZ DE VECINDARIO 3x3 EN TONOS DE GRIS:
    (El píxel objetivo central está resaltado entre corchetes)

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

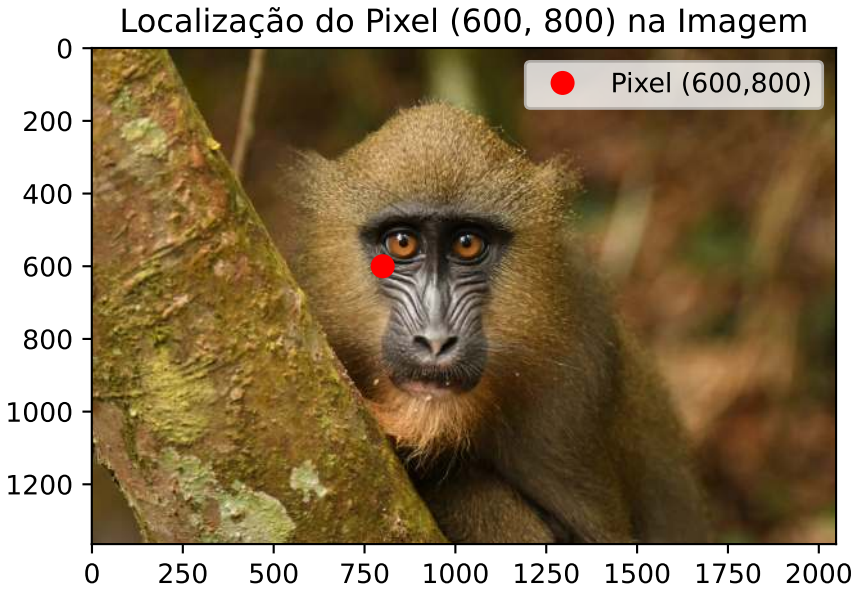


Figura 1.10: Acceso a un píxel específico (r, c) y la representación de su vecindario en la imagen.

Explicación línea por línea:

1. `img_gray[r, c]` — Devuelve un único valor entero (escalar) entre 0 y 255, que representa la intensidad de brillo del gris.
2. `img[r, c]` — Devuelve un vector con tres valores `[R, G, B]`, correspondientes a las intensidades de los canales Rojo, Verde y Azul.
3. `img_gray[r-1:r+2, c-1:c+2]` — Realiza un cortado (*slicing*) para extraer la submatriz 3×3 alrededor del píxel. Esta operación es la base para la implementación de filtros espaciales y convoluciones.
4. El bucle `for` subsiguiente solo formatea esa submatriz en la consola, mostrando el píxel central entre corchetes `[ ]` con fines didácticos.
5. La función `plt.plot(c, r, 'ro')` traza un punto rojo sobre la imagen para correlacionar la coordenada numérica con su posición visual. Nótese que Matplotlib utiliza el orden predeterminado de pantalla `(X, Y)`, invirtiéndolo a `(c, r)`.

Dado que la indexación en Python comienza en cero (*zero-based*), la esquina superior izquierda de la imagen es la coordenada `(0,0)`. Recuerde siempre esta regla: la primera dimensión de la matriz controla la **altura** (filas/Y) y la segunda controla la **anchura** (columnas/X).

1.14 Resumen

En este capítulo se presentaron los fundamentos de la representación de imágenes digitales: la definición de píxel, la estructuración de imágenes en matrices y el impacto del muestreo y la cuantificación en la calidad final:

- **Imagen digital** = función $f(x, y)$ que mapea coordenadas a intensidades (escalares o vectoriales).
- **Dominio**: conjunto finito $\mathbb{E} = \{(x, y) \in \mathbb{Z}^2 \mid 0 \leq x < L,\ 0 \leq y < H\}$.
- **Tipos principales**: binaria ($\mathcal{V} = \{0, 255\}$), escala de grises ($\mathcal{V} = [0, 255]$) y RGB ($\mathcal{V} = [0, 255]^3$).
- **Umbralización** convierte la escala de grises en binaria; el **método de Otsu** determina el valor de corte automáticamente maximizando la varianza entre clases.
- La biblioteca `morph.py` (o `mm`) ofrece funciones didácticas para operaciones básicas de PDI, como `mm.gray()`, `mm.threshold()`, `mm.show_multiple()`.
- **Trampa NumPy**: nunca uses `[[0]*n]*m` para crear matrices — usa siempre `np.zeros()` o `np.ones()`.

- Acceso a píxeles mediante `img[linha, coluna]`, con indexación ***zero-based***.

El Capítulo 2 abordará **histogramas y ecualización de contraste**.

1.15 🤖 Uso del Gemini Notebook como Tutor Complementario

En esta edición, además de los *notebooks* interactivos en Google Colab, el **Gemini Notebook** está disponible como herramienta complementaria de estudio. La plataforma utiliza exclusivamente los documentos proporcionados por el autor como base de conocimiento, garantizando respuestas coherentes con el contenido del libro.

🎓 Estudia con el Tutor Inteligente

Accede al entorno del capítulo a través del *enlace* a continuación y explora especialmente las opciones **Guía de Estudio** y **Conversación** para profundizar tu comprensión.
🚀 ACCEDER A GEMINI NOTEBOOK: CAPÍTULO 01

🌐 **Idioma y Lenguaje de Programación**

El proyecto de este capítulo en Gemini Notebook fue construido únicamente con el texto en **portugués** y los ejemplos de código en **Python**. Si estás estudiando con la edición en inglés o francés, o siguiendo la ruta en C++, las respuestas del tutor pueden no corresponder exactamente a la versión que estás leyendo.

⚠️ **Aviso sobre Contenido Generado por IA**

La IA es una poderosa aliada en los estudios, pero el contenido generado puede contener **errores o imprecisiones**. Consulta siempre **libros, artículos científicos y otras fuentes académicas confiables** para validar la información. Siempre que sea posible, ejecuta los ejemplos prácticos proporcionados en este capítulo para verificar los resultados.

Funcionalidades Disponibles en la Plataforma

Gemini Notebook ofrece una suite avanzada de herramientas basadas en IA para transformar el contenido estático del libro en una experiencia de aprendizaje dinámica y multimedia. La plataforma utiliza técnicas de **RAG** (*Retrieval-Augmented Generation*), fundamentadas en el trabajo de Lewis (2020), para basar las respuestas estrictamente en los documentos proporcionados y minimizar la ocurrencia de alucinaciones.

Las principales funcionalidades incluyen:

- **Resúmenes Multimodales (Audio y Video)**: Generación de conversaciones naturales entre expertos en el formato de **Resumen de Audio** (estilo *podcast*) y **Resumen de Video**, discutiendo los temas centrales del capítulo, como las diferencias entre PDI y VC, o la interpretación de transformaciones como la umbralización y el método de Otsu.
- **Visualización de Estructuras (Mapa Mental e Infografía)**: Creación automática de diagramas que conectan visualmente los conceptos, por ejemplo, el flujo de procesamiento desde la captura de la imagen digital, pasando por la conversión a tonos de gris, umbralización y segmentación binaria.
- **Herramientas de Evaluación (Prueba y Tarjetas Didácticas)**: Generación de **Pruebas** de opción múltiple y **Tarjetas Didácticas** (*flashcards*) para la fijación de conocimientos, basadas en el texto autoral (ej.: preguntas sobre la fórmula de conversión RGB→gris o sobre el funcionamiento del umbral global y de Otsu).
- **Apoyo a la Presentación (*Diapositivas* e Informes)**: Ayuda en la estructuración de **Presentaciones de *Diapositivas*** y en la redacción de **Informes** técnicos, facilitando la comunicación de resultados de experimentos con imágenes.
- **Análisis de Datos (Tabla de Datos)**: Organización de datos extraídos del texto en tablas

estructuradas, ayudando en la comprensión de ejemplos prácticos, como la comparación entre diferentes valores de umbral.

- ***Chat* Contextualizado**: Permite el cuestionamiento directo sobre el código y la teoría, como: *"¿Cómo implementar la conversión de RGB a tonos de gris usando los pesos del estándar ITU-R BT.601?"* o *"¿Qué sucede con la imagen binaria si elijo un umbral T=200 en lugar de T=128?"*.

1.16 Lista de Ejercicios

1. (15%) Con sus propias palabras, defina **imagen digital** y **píxel**. Dé un ejemplo concreto de cómo una imagen a color (RGB) se representa matricialmente en el computador.

2. (15%) Explique las diferencias entre **imagen binaria**, **escala de grises (8 bits)** y **color RGB**, indicando el rango de valores posibles para cada píxel en cada tipo.

3. (20%) Considerando la fórmula de conversión RGB → escala de grises del estándar ITU-R BT.601:

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

   Calcule el valor del píxel en escala de grises para $(R, G, B) = (80, 180, 30)$. Redondee al entero más cercano.

4. (20%) ¿Qué es **umbralización** (*thresholding*)? Explique la diferencia entre elegir un umbral T fijo (ej.: $T = 128$) y utilizar el **método de Otsu** para la determinación automática del umbral. En pocas palabras, ¿cómo elige el umbral el método de Otsu?

5. (15%) En el contexto de la biblioteca didáctica `mm` discutida en el capítulo, responda: - a) (7,5%) ¿Cómo se accede al valor del píxel en la posición (fila=50, columna=60) de una imagen en tonos de gris `img_gray`? - b) (7,5%) ¿Cuál es la ventaja de usar `mm.threshold(img_gray)` sin pasar el umbral? Compare con la llamada equivalente en OpenCV.

6. (15%) ¿Qué devuelve la propiedad `img.shape` para una imagen NumPy en formato RGB? Dé un ejemplo concreto con una imagen de 640×480 píxeles.

Referencias del Capítulo

La fundamentación teórica de este capítulo comprende las siguientes obras de PDI y VC:

- Gonzalez (2018) para los fundamentos de **Procesamiento Digital de Imágenes** (PDI).
- Singh (2019) para la implementación práctica de métodos de **procesamiento y análisis de imágenes**.
- Szeliski (2022) para el estudio de VC y algoritmos fundamentales.
- Bradski (2008) para la aplicación de la biblioteca **OpenCV** en el entorno Python.
- Lewis (2020) para el concepto de **generación aumentada por recuperación (RAG)**, utilizado en el soporte al procesamiento de la información de este material.

Executar Colab | Abrir GitHub

1.17 💻 Parte Práctica con Ejercicios de Programación

🎯 Objetivo de este Cuaderno

Los **Ejercicios de Programación (EPs)** presentados a continuación también pueden enviarse en actividades del Moodle (actividades VPL) que proporcionan *feedback* automático.

Este cuaderno fue desarrollado para superar limitaciones de uso del Moodle. Con él, se debe:

1. **Desarrollar:** Escribir y editar tu solución directamente en el entorno Colab.

2. **Validar:** Probar tu código localmente utilizando los **mismos casos de prueba** del Moodle.
3. **Organizar:** Guardar tus códigos de las actividades VPL de forma segura.
4. **Evaluar:** Cuando estés conectado al Moodle, basta con copiar tu solución y hacer clic en **Evaluar** en el Moodle (**si estás en la red de la UFABC**) para registrar tu nota oficial.

§ Instrucciones Paso a Paso

En un entorno de ejecución (como VSCode, Jupyter o Colab), siga el orden a continuación para configurar el entorno y validar sus ejercicios:

Preparación del Entorno

Ejecute la celda de código a continuación para descargar `morph.py` y `testsuite.py` del repositorio de la disciplina — solo si aún no existen en el directorio local. Con ambos en `./`, el *notebook* y los subprocesos del `TestSuite` encuentran el módulo sin configuraciones adicionales de ruta.

Nota: El script `testsuite.py` buscará automáticamente los casos de prueba en `all/{cap}/cases` en GitHub.

Escribiendo el Código

Guarde su solución en una celda de código usando el comando mágico `%%writefile`. El nombre del archivo debe seguir el patrón `EPX_Y.*`, donde `X` es el capítulo, `Y` es el ejercicio y `*` es la extensión del lenguaje.

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

Descarga

Descargue `morph.py` y `testsuite.py` ejecutando la celda a continuación:

```
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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Ejecutando los Testes

Después de guardar el archivo con tu solución, ejecuta el comando a continuación (en una nueva celda) para evaluar los testes automáticos:

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

Reemplaza la extensión según el lenguaje usado:

Lenguaje	Extensión
Python	`.py`
Java	`.java`
C	`.c`
C++	`.cpp`
JavaScript	`.js`

Lenguaje	Extensión
R	`.r`

Cómo funciona: El `TestSuite` descarga los casos de prueba de GitHub, ejecuta tu programa con cada entrada y compara la salida con la esperada – calculando automáticamente tu nota.

Para probar código Python directamente, sin guardar archivo, usa `run_code(codigo)` pasando el código como *string* en una variable `codigo`:

```
codigo = """
from morph import mm
# ... tu código aquí ...
"""
TestSuite("EP04_01").run_code(codigo)
```

⚠️ Importante: Reglas y Buenas Prácticas

🔹 Sobre la Entrada de Datos

Su programa debe leer la entrada estándar (teclado).

- **Python:** Utilice `input()`.
- **Otros lenguajes:** Utilice el comando de lectura estándar equivalente (`cin`, `Scanner`, etc.).

🔹 Configuración de IA en Colab

Para un mejor aprendizaje, se recomienda desactivar el autocompletado de código por IA, ya que no estará disponible durante las evaluaciones. Por ejemplo, en el navegador Chrome:

- Vaya a: **Herramientas > Configuración > IA generativa**
- Desmarque: **Habilitar generación de código**

🔹 Integridad Académica (Plagio)

Este recurso de pruebas locales se aplica a EPs **sin variaciones**. Sin embargo:

- **Individualidad:** Cada estudiante debe desarrollar su propia solución.
- **Detección de Similitud:** El profesor utiliza herramientas que detectan copias, **incluso con cambios de nombres de variables o espacios en blanco**.

1.17.1 EP01_01 📏 Tres métricas de distancia en PDI

En esta actividad, debes escribir un programa que calcule las tres distancias clásicas en PDI: **Euclidiana (L2)**, **City-Block (L1)** y ***Chessboard*** **(L∞)**.

- Lee **4 números reales** que representan las coordenadas: A_x, A_y, B_x, B_y.
- Calcula las tres distancias utilizando las fórmulas:

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

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

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

- Imprime los tres resultados, cada uno en una línea, formateados con **dos cifras decimales**, en el orden: **Euclidiana**, ***City-block***, ***Chessboard***.

📌 **Importante**:

- Utiliza las funciones matemáticas estándar de tu lenguaje: `math.sqrt`, `abs` (o `fabs`) y `max`.
- La salida debe contener **solo los números** (uno por línea), sin textos adicionales.
- Consulta un simulador interactivo para esta cuestión en la **Figura 1.11** (gráfico con arrastre de los puntos y visualización de las tres métricas).

1.17.1.1 🖼️ ¿Por qué es importante? – Costo computacional

En una imagen **1000×1000 píxeles** (1 millón de píxeles), calcular la distancia de cada píxel a un punto de referencia exige **1 millón de operaciones**. La elección de la métrica afecta el rendimiento:

Métrica	Operaciones por píxel	Costo relativo (1M píxeles)	Cuándo usar
Euclidiana (L2)	2 restas, 2 multiplicaciones, 1 suma, 1 `sqrt`	🔴 Más costosa – `sqrt` es cara	Distancia "real" en el espacio continuo
***City-block* (L1)**	2 restas, 2 `abs`, 1 suma	🟡 Moderada – sin raíz cuadrada	Cuadrículas, robótica, imágenes binarias
***Chessboard* (L∞)**	2 restas, 2 `abs`, 1 `max`	🟢 Más eficiente	Movimientos de piezas, morfología

La función `sqrt` es computacionalmente más cara que operaciones como suma, resta, multiplicación y valor absoluto. En CPUs modernas, la diferencia puede ser pequeña (alrededor de 1,5× a 3×), pero en sistemas embebidos o en bucles de millones de iteraciones, cualquier ganancia importa. Por eso, **cuando el objetivo es solo comparar distancias** (p. ej.: encontrar el punto más cercano), usa la **distancia euclidiana al cuadrado**.

1.17.1.2 📋 Tarea (especificación para VPL)

Entrada:
Una única línea con cuatro números reales: `Ax Ay Bx By`

Salida:
Tres líneas, cada una con un número real de dos cifras decimales (Euclidiana, City-block, Chessboard).

1.17.1.3 📌 Ejemplos

Entrada	Salida	Observación

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

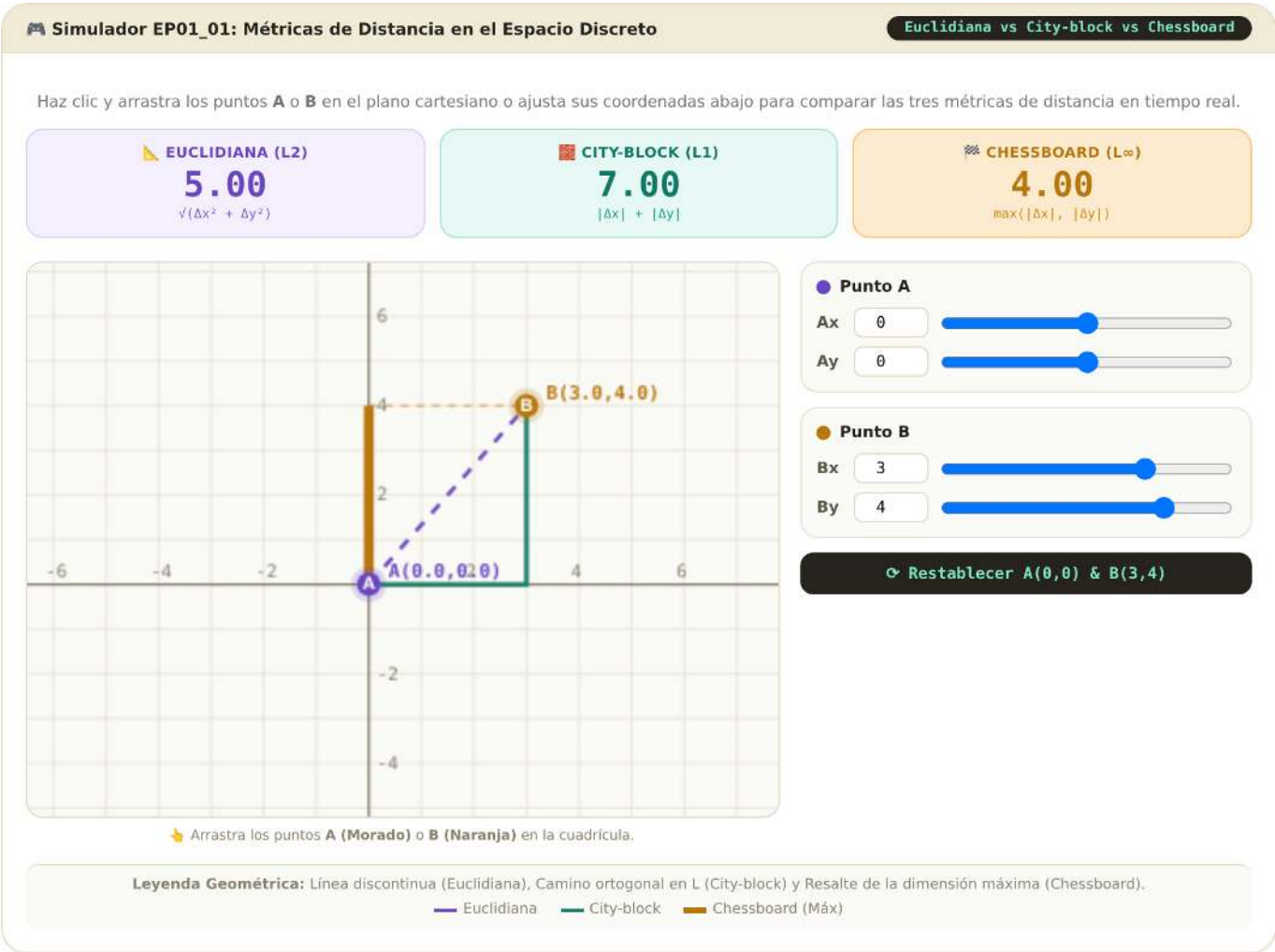
`0`
`0`
`1`
`1`
	`1.41`
	`2.00`
	`1.00`
		Diagonal unitaria

Ejemplo de prueba de `sqrt` en Python, con `timeit` aislando cada operación:

```
import math
import timeit
```

```python
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"Suma simple       : {t_soma:.3f} s")
print(f"Suma + sqrt        : {t_sqrt:.3f} s")
print(f"Razón (sqrt/suma)  : {t_sqrt/t_soma:.2f}x")
```

```
Suma simple       : 3.242 s
Suma + sqrt        : 5.445 s
Razón (sqrt/suma)  : 1.68x
```



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

Figura 1.11: Simulador EP01_01: Distancias Euclidiana, City-block y Chessboard

1.17.1.4 🐍 Python

Basta crear una celda de código normal e insertar el código Python. La entrada se puede simular usando `input()`, que funciona normalmente.

Ejemplo de celda:

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

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

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

# 3. Distancia Chebyshev / Tablero de ajedrez (Linf)
dist_chessboard = max(dx, dy)

# Salida formateada según los casos de prueba
print(f"{dist_euclidiana:.2f}")
print(f"{dist_city_block:.2f}")
print(f"{dist_chessboard:.2f}")
```

```
Overwriting EP01_01.py
```

```
# Espera que usted escriba 4 números enteros al ejecutar esta celda.
# En Jupyter o Google Colab, la magia %run -i permite que el script lea del teclado.
# En la terminal común, usted usaría: python3 EP01_01.py (sin el '!' y sin '%run').

# %run -i EP01_01.py
```

```
# Envía 4 enteros como entrada estándar (stdin) al script EP01_01.py usando un pipe
!echo -e "0\n0\n4\n4" | python3 EP01_01.py
```

```
5.66
8.00
4.00
```

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

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

1.17.1.5 ☕ Java

Para ejecutar Java en Colab, debes usar una celda con el prefijo `%%writefile` para guardar el código en un archivo, compilar y ejecutar.

```
%%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>
```

1.17.1.6 💻 C

De forma similar, use `%%writefile` para guardar el código, luego compile y ejecute. Para C, utilizamos el compilador **GCC**.

```
# Instalar el compilador GCC y herramientas de build
# El build-essential incluye gcc, g++, make, etc.
import platform, shutil, subprocess

def instalar_gcc():
    if shutil.which("gcc"):
        print("  GCC ya disponible."); return
    if platform.system() != "Linux":
        print("  Mac: xcode-select --install | Windows: WSL o 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 instalado!")

instalar_gcc()
```

```
  GCC ya disponible.
```

```
%%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
```

```
# Compila el archivo .c generando el ejecutable EP01_01
# -lm se usa para enlazar la biblioteca matemática (math.h) si es necesario

!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>
```

1.17.1.7 💻 C++

De manera similar a Java, usa `%%writefile` para guardar el código, luego compila y ejecuta. Recuerda que, en Colab, también es necesario instalar lo siguiente:

```
# Instalar el compilador G++ para C++
# El build-essential incluye el g++, make, etc.
import platform, shutil, subprocess

def instalar_gpp():
    if shutil.which("g++"):
        print("  G++ ya disponible."); return
    if platform.system() != "Linux":
        if platform.system() == "Darwin":
            print("  Mac: xcode-select --install")
        else:
            print("  Windows: use WSL o 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("  Compilador C++ listo.")

instalar_gpp()
```

```
  G++ ya disponible.
```

```
%%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>
```

1.17.1.8 🌐 JavaScript (Node.js)

Para JavaScript, use `%%writefile` para crear el archivo y ejecútalo con 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>
```

1.17.1.9 📊 R

En Colab, se puede ejecutar código R directamente utilizando la magia `%%R`.
El programa debe leer **cuatro números** (x1, y1, x2, y2) y mostrar la distancia euclidiana con **dos decimales**.

Ejemplo de celda:

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

```
# Instalar R y Rscript
# r-base instala el entorno R completo, incluido Rscript
import platform, shutil, subprocess

def instalar_r():
    if shutil.which("R"):
        print("  R ya disponible."); 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("  Entorno R listo.")

instalar_r()
```

```
  R ya disponible.
```

Para probar de la misma manera que en los ejemplos anteriores, se debe usar la entrada estándar (stdin) en la terminal o adaptar el código según lo siguiente:

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

# Salidas: Euclidiana, City-block y 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>
```

1.17.2 EP01_02 📊 Desempeño Predictivo — Métricas de ML en VC

En esta actividad, entrarás al mundo del **Aprendizaje Automático** (*Machine Learning*). Tu objetivo es evaluar el desempeño de un clasificador binario calculando métricas a partir de una **Matriz de Confusión**.

1.17.2.1 🧠 ¿Por qué importa la métrica correcta?

Imagina un detector de **monedas de R$ 1,00**. El impacto del error define la métrica prioritaria:

Métrica	Ejemplo Práctico	Importancia en PDI/VC
Exactitud	**Conteo de Granos**	Útil cuando las clases están equilibradas (ej: la mitad de los granos con defecto, la mitad sanos).
Precisión	**Seguridad/Biometría**	Crucial para evitar **Falsos Positivos** (ej: no permitir que un impostor acceda a un sistema por error de reconocimiento).
Sensibilidad	**Salud (Tumores)**	Crucial para evitar **Falsos Negativos** (ej: no dejar que un tumor pase desapercibido en un examen de Rayos-X).
Puntuación F1	**Billetes de Dinero**	Ideal para un **equilibrio** entre no rechazar billetes verdaderos y no aceptar billetes falsos.

1.17.2.2 📊 La Matriz de Confusión

	Predicha Positiva	**Predicha Negativa**
Real Positiva	**VP** (Verdadero Positivo)	**FN** (Falso Negativo)
Real Negativa	**FP** (Falso Positivo)	**VN** (Verdadero Negativo)

Tarea:

1. Lee **4 valores enteros** en el orden: `VP`, `FN`, `FP`, `VN`.
2. Calcula las métricas utilizando las fórmulas:

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

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

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

$$\text{Puntuación F1} = \frac{2 \times \text{Precisión} \times \text{Sensibilidad}}{\text{Precisión} + \text{Sensibilidad}}$$

3. Imprime los resultados formateados con **dos decimales**, uno por línea.

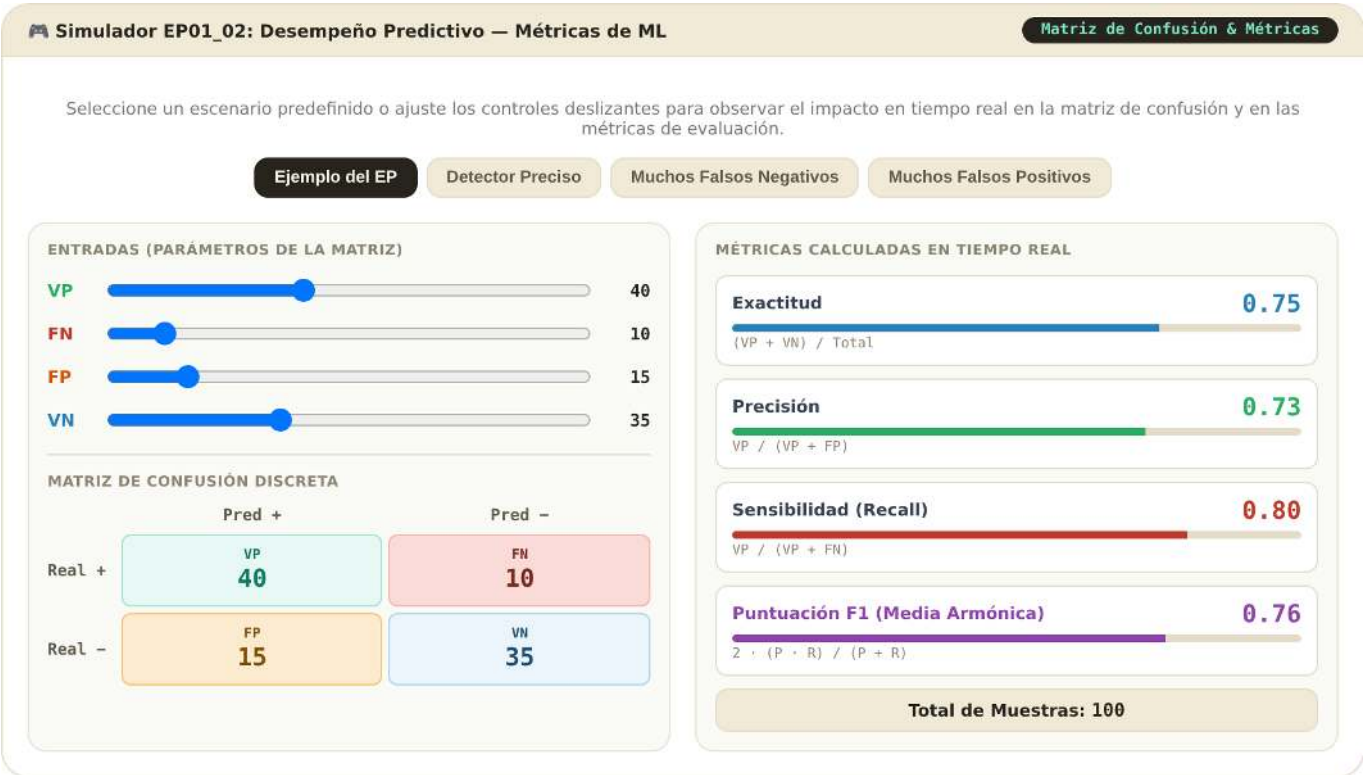
📌 **Importante**:

- Usa división en punto flotante para evitar resultados truncados.
- El orden de salida debe ser: **Exactitud**, **Precisión**, **Sensibilidad** y **Puntuación F1**.
- Consulta la simulación interactiva de este EP en la Figura 1.12.

1.17.2.3 📌 Ejemplo de Ejecución

Entrada	Salida	Observación
40	0.75	Exactitud
10	0.73	Precisión
15	0.80	Sensibilidad
35	0.76	Puntuación F1

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



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

Figura 1.12: Simulador EP01_02: Rendimiento Predictivo — Métricas de ML

```
%%writefile EP01_02.py
# tu solución
```

```
Overwriting EP01_02.py
```

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

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

1.17.3 EP01_03 ↗ *Mean Average Precision* (mAP) — Curva Precisión-Sensibilidad

En esta actividad, evaluará un clasificador binario (p. ej., detección de deforestación en imágenes de satélite, ver dgi.inpe.br) mediante la **curva Precisión-Sensibilidad** y la métrica **mAP** (***Mean Average Precision***).

El mAP es estándar en competencias como COCO (*Common Objects in Context*) y PASCAL VOC (*Visual Object Classes*) y de los modelos YOLO (*You Only Look Once*).

1.17.3.1 🧠 ¿Por qué el mAP es la métrica-estándar?

En la EP01_02 vio que la elección del umbral altera significativamente la Precisión y la Sensibilidad. El **mAP (*Mean Average Precision*)** resuelve esto: evalúa el modelo en **varios umbrales** (cada umbral debe generar una matriz de confusión diferente) y resume el desempeño mediante el **área bajo la curva Precisión-Sensibilidad (P-S)**.

Mientras que el F1-*Score* observa un único punto de equilibrio, el mAP considera la curva completa. Cuanto más cercano a **1,0**, mejor es el detector en todos los umbrales y clases (p. ej., monedas de 25, 50 y 1 real).

Métrica	Qué resume	Limitación
F1-*Score*	Equilibrio P × S en un único umbral	Depende del umbral elegido
AP	Área bajo la curva P-S de una clase	Válida solo para una clase
mAP	Promedio de las APs sobre todas las clases	Más complejo de implementar

Referencias: Roboflow — mAP · Vídeo explicativo

1.17.3.2 Cómo se calcula el mAP — paso a paso

1. **Umbrales fijos** (use siempre esta lista):

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

2. Para cada umbral (t), clasifique las muestras: `predicho = 1 si confianza  t, sino 0`. Calcule VP, FP, FN, VN y obtenga Precisión((t)) y Sensibilidad((t)).
3. Construya la curva P-S: pares (Sensibilidad((t)), Precisión((t))), ordenados por Sensibilidad creciente.
4. **Monotonice la Precisión**:

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

5. Calcule la **AP** (área bajo la curva monotónica) usando la **regla del trapecio** (aproximación más precisa que la simple suma de Riemann):

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

6. **mAP** = promedio de las APs de todas las clases. En este EP hay solo **1 clase**, por lo tanto mAP = AP.

> **Nota**
>
> 📐 **Diferencia resumida:**
> La *suma de Riemann* aproxima el área mediante rectángulos, pudiendo subestimar o sobrestimar. La **regla del trapecio** usa trapecios, reduciendo el error al considerar el promedio entre los valores en los extremos del intervalo, siendo generalmente más precisa para funciones suaves por partes, como la curva Precisión-Sensibilidad.

1.17.3.3 📋 Tarea

Lea un entero `n` (cantidad de muestras). Luego lea `n` líneas, cada una con: `verdad` (0 o 1) y `confianza` (float 0.0–1.0).

Calcule e imprima, para el **umbral 0.85** (índice 9 de la lista):

- Matriz de Confusión (VP, FN, FP, VN)
- Exactitud, Precisión, Sensibilidad y F1-Score

A continuación, para **todos los umbrales**, imprima:

- Precisiones crudas, Precisiones monotónicas y Sensibilidades, separadas por `,`
- mAP final

1.17.3.4 📌 Importante

- Umbral fijo para las métricas individuales: **0.85**
- División segura: si el denominador es cero, use 0
- Formato: dos decimales
- Monotonice de atrás hacia adelante
- La Figura 1.13 presenta una simulación de esta cuestión

1.17.3.5 📌 Ejemplo de Ejecución

Entrada	Salida Esperada
7 0 0.94 1 0.80 1 0.69 0 0.67 1 0.30 1 0.15 1 0.15	# MÉTRICAS PARA EL UMBRAL 0.85 # Matriz de Confusión: VP = 0, FN = 5 FP = 1, VN = 1 Métricas de Evaluación: Exactitud: 0.14 Precisión: 0.00 Sensibilidad: 0.00 F1-Score: 0.00 # MÉTRICAS PARA TODOS LOS UMBRALES # Precisiones: 0.00, 0.00, 0.00, 0.50, 0.50, 0.50, 0.50, 0.50, 0.60, 0.71, 0.71 Precisiones mon.: 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71 Sensibilidades: 0.00, 0.00, 0.00, 0.20, 0.40, 0.40, 0.40, 0.40, 0.60, 1.00, 1.00 mAP: 0.71

1.17.3.6 🐍 Consejo para calcular el AP (con regla del trapecio)

```python
def calcular_AP(verdades, confianzas, umbrales):
    m = len(umbrales)
    precisiones = [0.0] * m
    sensibilidades = [0.0] * m
    for i in range(m):
        p, s = calcular_metricas(verdades, confianzas, umbrales[i])
        precisiones[m-1-i] = p
        sensibilidades[m-1-i] = s
```

```
    prec_mono = precisiones.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):
        # Regla del trapecio: promedio de las alturas por la base
        area_trapecio = (prec_mono[i-1] + prec_mono[i]) / 2.0
        AP += area_trapecio * (sensibilidades[i] - sensibilidades[i-1])
    return precisiones, prec_mono, sensibilidades, AP
```

```
%%writefile EP01_03.py
# tu solución
```

```
Overwriting EP01_03.py
```

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

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

1.17.4 EP01_04 🖼️ Lectura e Información de una Imagen en Matriz

En esta actividad, debes escribir un programa que procese una imagen digital representada como una matriz de píxeles en tonos de gris.

- Lee dos enteros **L** y **C**, que representan el número de filas y columnas.
- Lee los **L * C** valores enteros que componen la matriz de la imagen (cada valor entre 0 y 255).
- Calcula e imprime la siguiente información:

1. El número de filas.
2. El número de columnas.
3. El valor del píxel más grande (**Máximo**).
4. El valor del píxel más pequeño (**Mínimo**).
5. La **Media** aritmética de todos los píxeles.

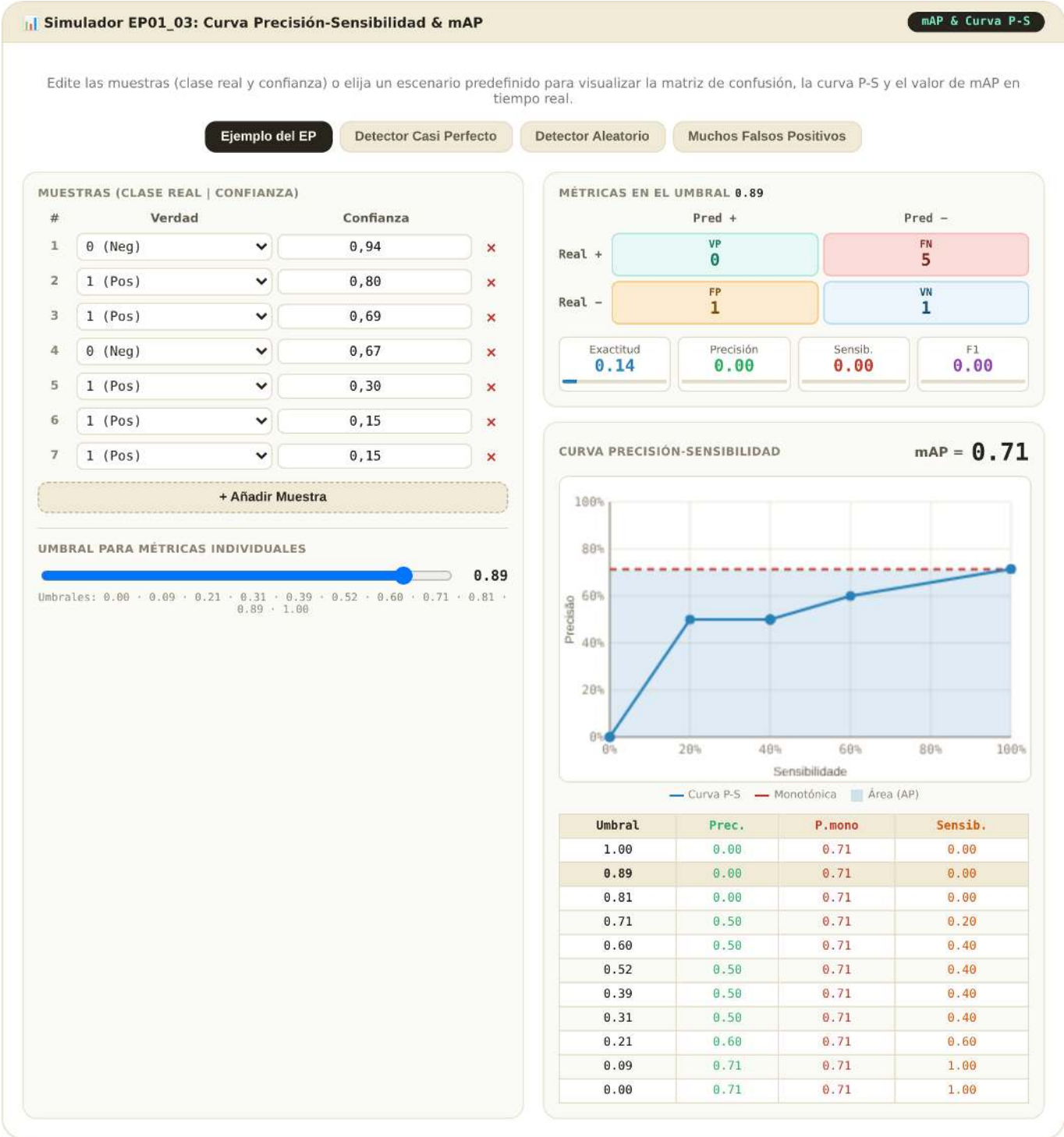
📌 **Importante**:

- La salida debe seguir exactamente el formato etiquetado (ej.: `Filas: X`).
- El valor de la media debe formatearse con **dos decimales**.
- Consulta un simulador interactivo para esta pregunta en la **Figura 1.14** (cuadrícula interactiva para visualización de intensidades y cálculos en tiempo real).

1.17.4.1 🧠 ¿Por qué es importante? – La Imagen como Datos

Toda imagen digital es, en el fondo, una estructura de datos. En tonos de gris de 8 bits, cada píxel es un valor escalar. Extraer estadísticas básicas es el primer paso para:

Operación	Utilidad Práctica
Máximo/Mínimo	Identificar si la imagen está "lavada" (bajo contraste) o saturada.
Media	Calcular el brillo global de la escena para ajustes de exposición.
Normalización	Redimensionar los valores a intervalos como $[0, 1]$ en redes neuronales.



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

Figura 1.13: Simulador EP01_03: Precisión Media Promedio (mAP) y Curva P-S

1.17.4.2 📋 Tarea (especificación para VPL)

Entrada:

La primera línea contiene el entero `L` (filas).

La segunda línea contiene el entero `C` (columnas).

Las líneas siguientes contienen los elementos de la matriz.

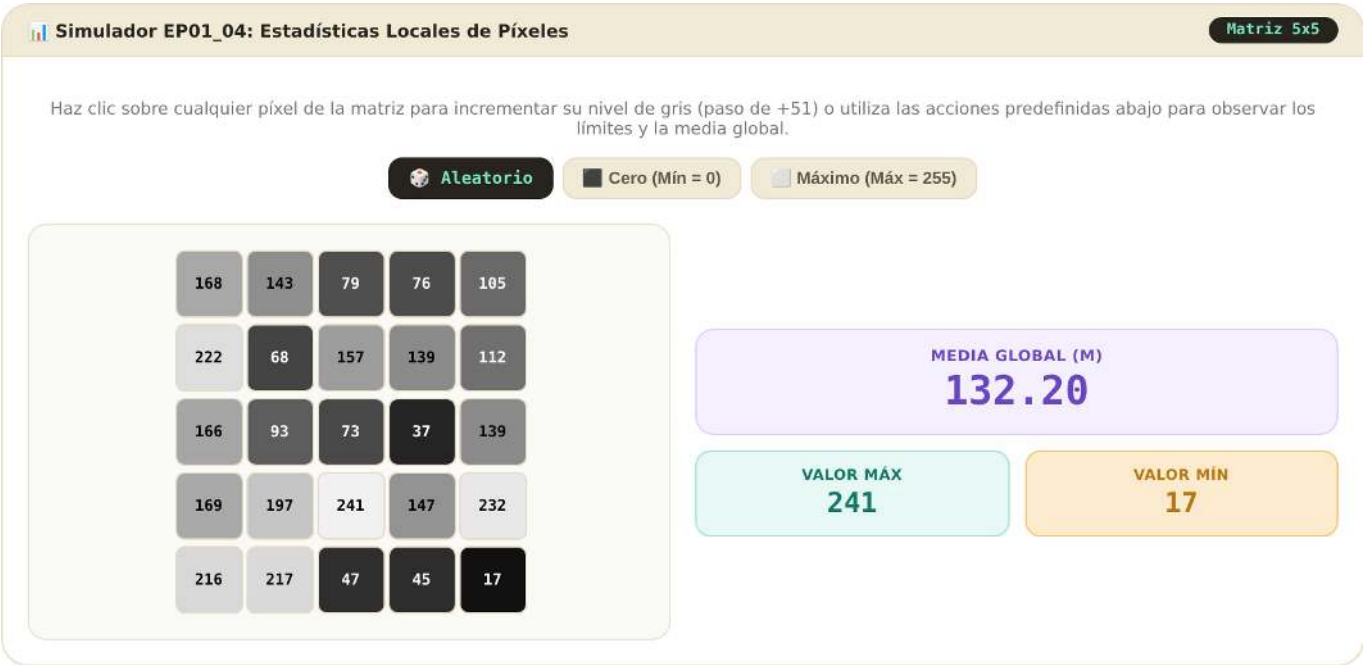
Salida:

Cinco líneas formateadas según el ejemplo:

```
Filas: L
Columnas: C
Max: V
Min: V
Media: V.VV
```

1.17.4.3 📌 Ejemplos

Entrada	Salida	Observación
2 3 0 128 255 50 100 200	Filas: 2 Columnas: 3 Max: 255 Min: 0 Media: 122.17	Imagen pequeña de alto contraste



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

Figura 1.14: Simulador EP01_04: Estadísticas de Píxeles en Matriz Discreta 5x5

```
%%writefile EP01_04.py
# tu solución
```

```
Overwriting EP01_04.py
```

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

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

1.17.5 EP01_05 🔄 Negativo de una Imagen en Tonos de Gris

En esta actividad, se debe escribir un programa que calcule el negativo de una imagen digital.

- Lea dos enteros **L** y **C**, que representan el número de filas y columnas.
- Lea los valores enteros que componen la matriz de la imagen.
- Para cada píxel, aplique la transformación de inversión:

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

- Imprima la matriz resultante, manteniendo el formato original (L filas y C columnas).

📌 **Importante**:

- Los valores de cada fila en la salida deben estar separados por un **espacio en blanco**.
- La salida debe contener **solo los números** de la matriz resultante.
- Consulte un simulador interactivo para esta cuestión en la **Figura 1.15** (comparación en tiempo real entre la matriz original y su negativo).

1.17.5.1 🧠 ¿Por qué es importante? – Inversión de Intensidad

El **negativo** es una transformación lineal básica que invierte la escala de brillo. Es una herramienta esencial para que el ojo humano identifique detalles claros que están "ocultos" en fondos más oscuros, siendo ampliamente utilizada en:

Aplicación	Utilidad
Imágenes Médicas	Mejora la visualización de anomalías en tejidos densos (ej: Rayos-X).
Astronomía	Destacar galaxias y nebulosas tenues contra el vacío del espacio.
Artes Digitales	Efectos estéticos y preparación de máscaras de selección.

1.17.5.2 📋 Tarea (especificación para VPL)

Entrada:

La primera línea contiene el entero `L`.

La segunda línea contiene el entero `C`.

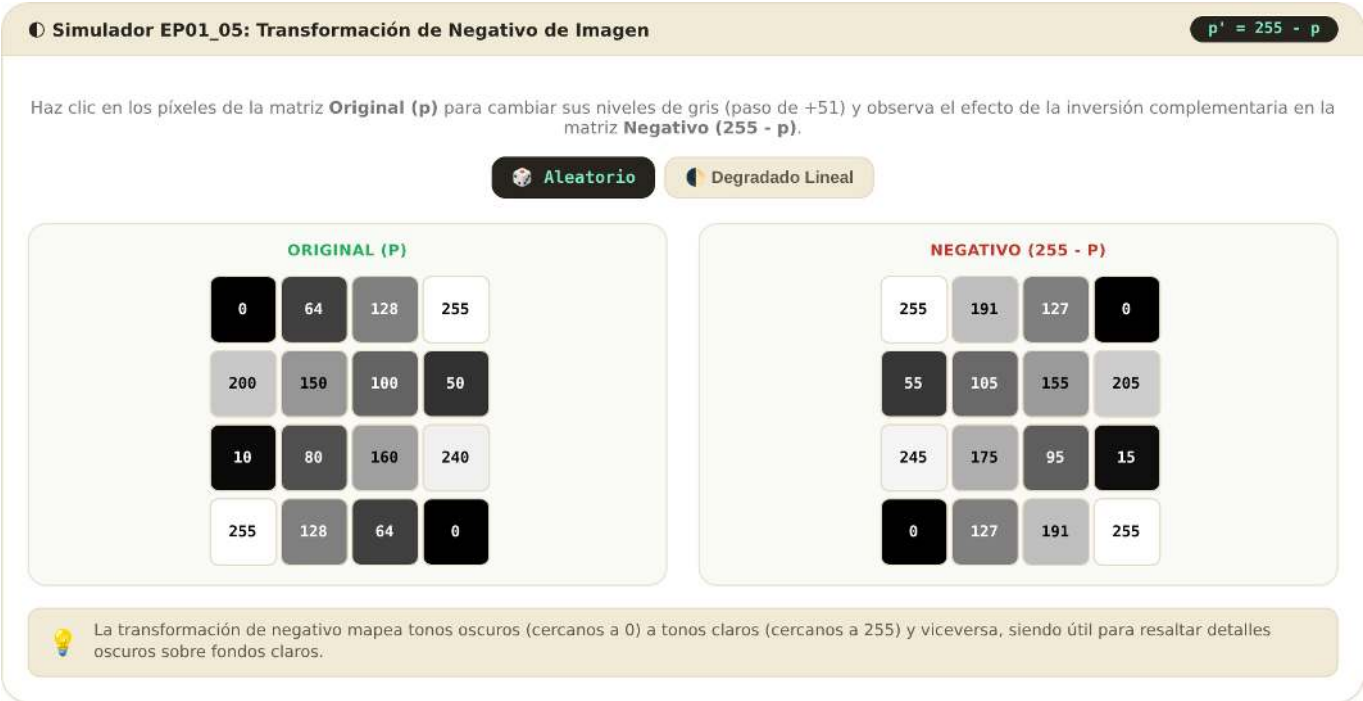
Las líneas siguientes contienen los elementos de la matriz.

Salida:

La matriz invertida con `L` filas y `C` columnas.

1.17.5.3 📌 Ejemplos

Entrada	Salida	Observación
2 3 0 128 255 50 100 255	255 127 0 205 155 55	Donde era 0 (negro) se convierte en 255 (blanco)



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

Figura 1.15: Simulador EP01_05: Transformación de Negativo de Imagen (Inversión de Intensidad)

```
%%writefile EP01_05.py
# tu solución
```

```
Overwriting EP01_05.py
```

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

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

1.17.6 EP01_06 🎨 Conversión RGB → Escala de Grises (ITU-R BT.601)

En esta actividad, debes escribir un programa que convierta píxeles de color (RGB) a escala de grises utilizando la ponderación fisiológica del estándar ITU-R BT.601.

- Lee dos enteros **L** y **C**, que representan el número de filas y columnas.
- Lee **L** × **C** tripletas de enteros, donde cada tripleta representa los canales **R** (Red), **G** (Green) y **B** (Blue) de un píxel.
- Para cada píxel, calcula el valor de gris (g) usando la fórmula:

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

- Imprime la matriz resultante (L filas y C columnas) que contenga los valores enteros convertidos.

📌 **Importante**:

- Utiliza la función `round()` de tu lenguaje para garantizar el redondeo correcto al entero más cercano.
- La salida debe contener únicamente los valores de gris, manteniendo la estructura de matriz (separados por espacios en la línea).
- Consulta un simulador interactivo para esta pregunta en la **Figura 1.16** (ajusta los deslizadores para ver cómo contribuye cada color al brillo final).

1.17.6.1 🧠 ¿Por qué no usar solo el promedio?

El ojo humano no percibe todos los colores con la misma intensidad. Somos mucho más sensibles al **Verde** que al **Azul** debido a nuestra evolución biológica. El estándar ITU-R BT.601 utiliza pesos específicos para crear una imagen en gris que parezca naturalmente correcta para nuestra visión:

Canal	Peso	Percepción Humana
🟢 **Verde**	**58.7%**	Máxima sensibilidad (distinción de follajes).
🔴 **Rojo**	**29.9%**	Sensibilidad media.
🔵 **Azul**	**11.4%**	Baja sensibilidad (tonos más oscuros).

1.17.6.2 📋 Tarea (especificación para VPL)

Entrada:

La primera línea contiene el entero `L`.

La segunda línea contiene el entero `C`.

Las líneas siguientes contienen tripletas de enteros `R G B` para cada píxel.

Salida:

La matriz de grises con `L` filas y `C` columnas.

1.17.6.3 📌 Ejemplos

Entrada	Salida	Observación
1 3 255 0 0 0 255 0 0 0 255	76 150 29	Observa cómo el Verde (150) es más brillante que el Azul (29)

```
%%writefile EP01_06.py
# tu solución
```

```
Overwriting EP01_06.py
```

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

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

1.17.7 EP01_07 ⚫ Umbralización Manual: Imagen Binaria

En esta actividad, debes escribir un programa que realice la segmentación de una imagen mediante la umbralización (*thresholding*).

- Lee dos enteros **L** y **C**, que representan las dimensiones de la matriz.

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

Figura 1.16: Simulador EP01_06: Conversión RGB a Escala de Grises (Ponderación Perceptual ITU-R BT.601)

- Lee un entero **T**, que será el valor del umbral (corte).
- Lee los valores enteros de la matriz.
- Para cada píxel p, aplica la siguiente regla de binarización:

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

- Imprime la matriz resultante que contenga solo los valores 0 o 255.

📌 **Importante**:

- Presta atención al operador: el píxel solo se vuelve blanco (255) si es **estrictamente mayor** que T.
- La salida debe mantener la estructura de matriz (L filas y C columnas).
- Consulta un simulador interactivo para esta pregunta en la **Figura 1.17** (ajusta el control deslizante de T para observar cómo los objetos se aíslan del fondo).

1.17.7.1 🧠 ¿Qué es la Segmentación?

La umbralización es el método más simple para separar objetos de interés del fondo de la imagen. Al transformar tonos de gris en blanco y negro puro, creamos un mapa binario que facilita el conteo de objetos o la identificación de formas:

Valor del Píxel (p)	Condición	Resultado Final
Oscuro ($p \leq T$)	Fondo/Ruido	**0** (Negro)
Claro ($p > T$)	Objeto/Destacado	**255** (Blanco)

1.17.7.2 📋 Tarea (especificación para VPL)

Entrada:

La primera línea contiene el entero `L`.

La segunda línea contiene el entero `C`.

La tercera línea contiene el entero `T` (umbral).

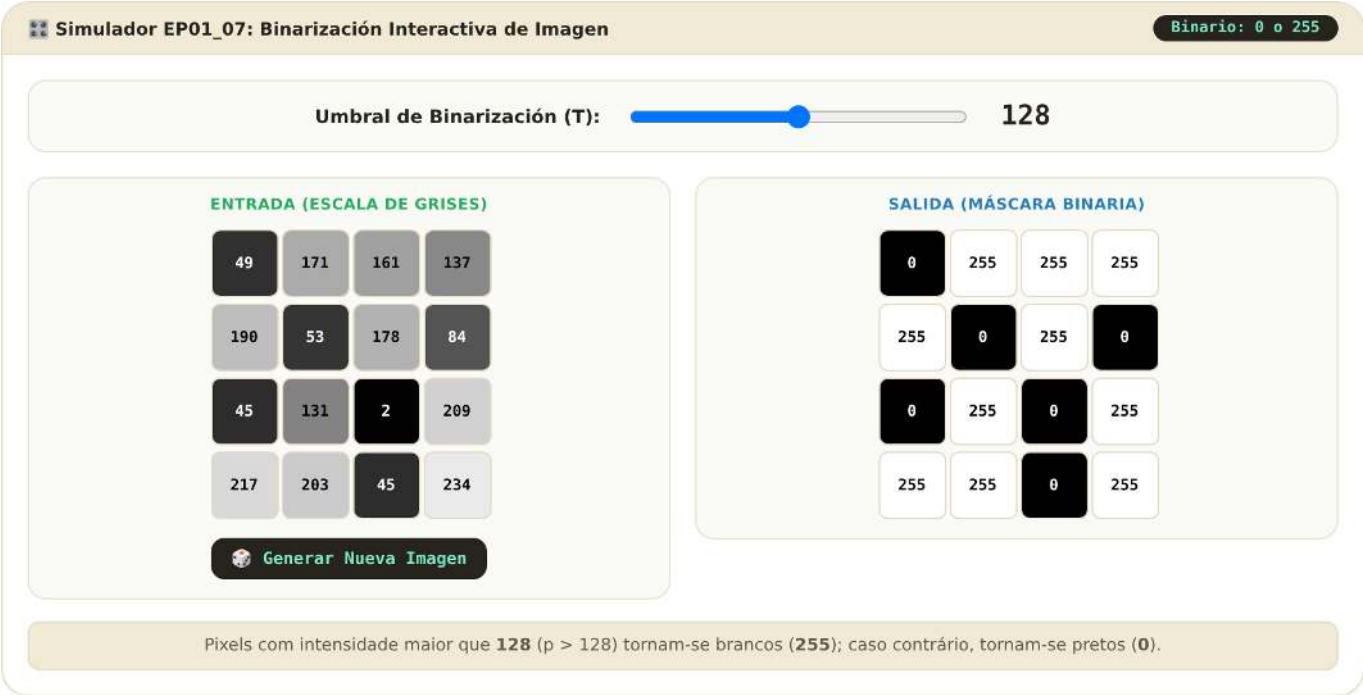
Las líneas siguientes contienen los elementos de la matriz.

Salida:

La matriz binarizada (0 o 255) con `L` filas y `C` columnas.

1.17.7.3 📌 Ejemplos

Entrada	Salida	Observación
2 4 128 0 100 128 200 50 129 255 64	0 0 0 255 0 255 255 0	Observa que el valor 128 se convirtió en 0 (pues 128 $\leq$ 128)



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

Figura 1.17: Simulador EP01_07: Umbralización Global Interactiva (Binarización p > T)

```
%%writefile EP01_07.py
# tu solución
```

```
Overwriting EP01_07.py
```

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

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

1.17.8 EP01_08 🎨 Remapeamento por Rango de Intensidad

En esta actividad, debes escribir un programa que aplique transformaciones lineales distintas a diferentes regiones de intensidad de la imagen.

- Lee dos enteros **L** y **C**, que representan las dimensiones de la matriz.
- Lee los enteros **T** (umbral), (delta 1) y (delta 2).

- Lee los valores enteros de la matriz.
- Para cada píxel p, aplica la regla de remapeo condicional:

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

- Imprime la matriz resultante con los nuevos valores de intensidad.

📌 **Importante**:

- es el offset aplicado a los píxeles oscuros (por debajo del umbral).
- es el offset aplicado a los píxeles claros (mayores o iguales al umbral).
- Los casos de prueba garantizan que el resultado siempre estará en el rango válido de **0 a 255**, por lo tanto, no es necesario manejar saturación ni redondeos.
- La salida debe mantener la estructura de matriz (**L** filas y **C** columnas).

1.17.8.1 🧠 Transformación Condicional de Píxeles

En Procesamiento Digital de Imágenes (PDI), con frecuencia necesitamos tratar regiones de forma independiente. Esta técnica permite, por ejemplo, aclarar solo las sombras de una fotografía (aumentando los píxeles oscuros) sin estallar el brillo de las áreas ya claras, o viceversa.

Rango de Intensidad	Condición	Operación
Píxeles Oscuros	$p < T$	$p + \delta_1$
Píxeles Claros	$p \geq T$	$p + \delta_2$

1.17.8.2 📋 Tarea (especificación para VPL)

Entrada:

La primera línea contiene los enteros `L` y `C`.

La segunda línea contiene los enteros `T`, y .

Las líneas siguientes contienen los elementos de la matriz.

Salida:

La matriz transformada con `L` filas y `C` columnas, con valores separados por espacios.

1.17.8.3 📌 Ejemplos

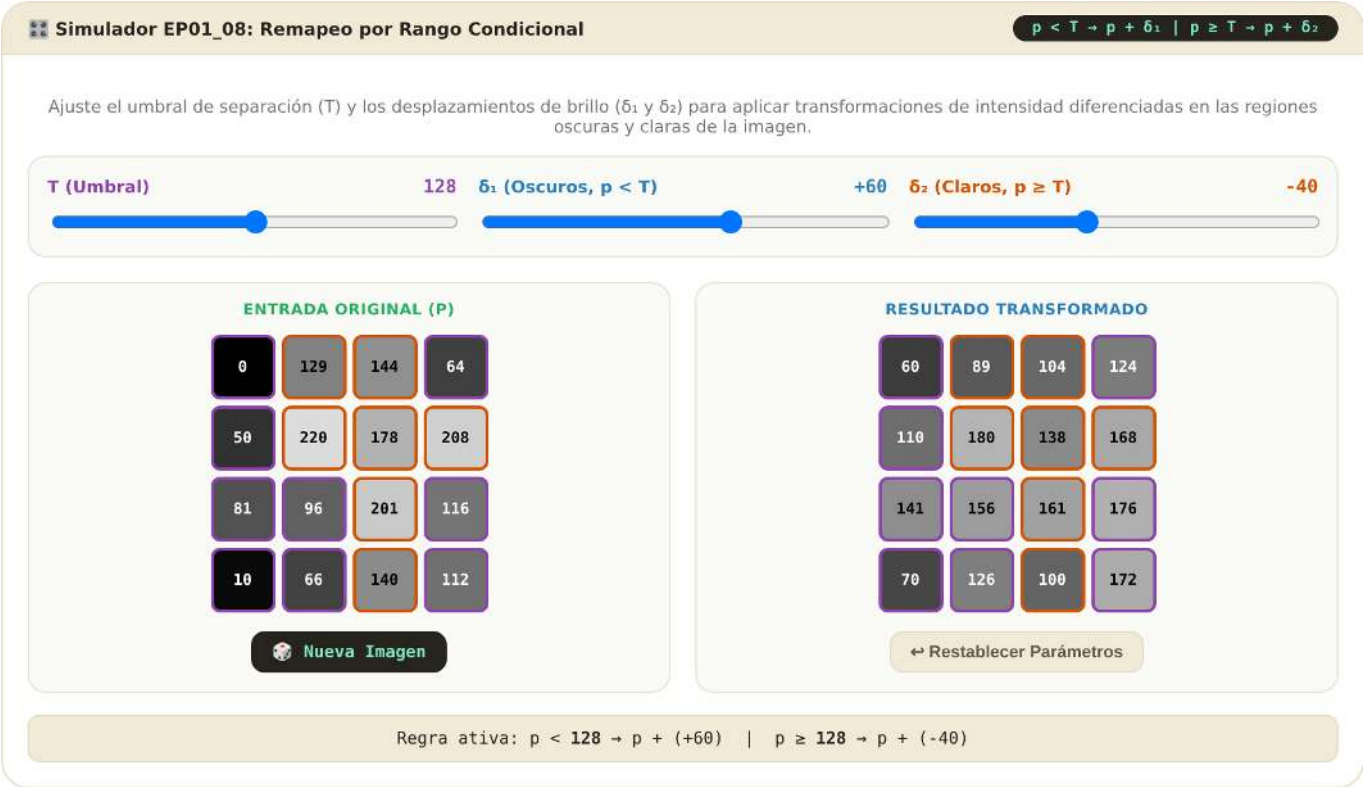
Entrada	Salida	Observación

`2 4`
`128 60 -40`
`0 100 150 255`
`80 128 200 30`
	`60 160 110 215`
	`140 88 160 90`
		Píxeles $<$ 128 suman 60. Píxeles 128 restan 40.

```
%%writefile EP01_08.py
# tu solución
```

```
Overwriting EP01_08.py
```

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



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

Figura 1.18: Simulador EP01_08: Remapeo por Rango Condicional (Brillo y Contraste por Umbral)

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

1.17.9 EP01_09 🏁 Patrón de Tablero: Matriz de Ajedrez

En esta actividad, debes escribir un programa que genere una imagen sintética con el patrón de un tablero de ajedrez.

- Lee dos enteros **L** (filas) y **C** (columnas).
- Genera una matriz donde los valores alternan entre **0** (negro) y **1** (blanco).
- La lógica de relleno debe seguir la regla de paridad:
- El elemento en la posición $(0, 0)$ siempre es **0**.
- Un píxel en la posición (i, j) será **1** si la suma de los índices $(i + j)$ es **impar**.
- Un píxel en la posición (i, j) será **0** si la suma de los índices $(i + j)$ es **par**.

📌 **Importante**:

- Los colores deben alternar correctamente tanto horizontal como verticalmente.
- La salida debe ser la matriz impresa línea por línea, con los elementos separados por un espacio.
- Consulta un simulador interactivo para esta cuestión en la **Figura 1.19** (ajusta las dimensiones para visualizar la construcción de la malla y la salida textual correspondiente).

1.17.9.1 🧠 Patrones Sintéticos

Crear patrones geométricos es un ejercicio fundamental para dominar la **lógica de índices** en matrices. En Procesamiento Digital de Imágenes, el patrón de ajedrez no solo es estético; también se utiliza ampliamente para:

Aplicación	Utilidad
Calibración de Cámara	Estimar parámetros intrínsecos y extrínsecos de la lente.
Corrección de Distorsión	Identificar y corregir el efecto de “barril” o “cojín” en lentes gran angulares.
Mapeo 3D	Proyectar patrones conocidos para reconstruir superficies en sistemas de luz estructurada.

1.17.9.2 📋 Tarea (especificación para VPL)

Entrada:

Una línea que contiene el entero `L` (filas).

Una línea que contiene el entero `C` (columnas).

Salida:

La matriz de ajedrez con `L` filas y `C` columnas, impresa con espacios entre los elementos.

1.17.9.3 📌 Ejemplos

Entrada	Salida	Observación
3 4	0 1 0 1 1 0 1 0 0 1 0 1	Observa que cada fila comienza con el inverso de la anterior

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

Figura 1.19: Simulador EP01_09: Generador de Patrón de Cuadros (Lógica de Paridad (i + j) % 2)

```
%%writefile EP01_09.py
# tu solución
```

```
Overwriting EP01_09.py
```

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

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

1.17.10 EP01_10 📄 Metadatos: Lectura de Archivo PGM

En esta actividad, debes leer un archivo de imagen en el formato **PGM (Portable Gray Map)** y extraer sus dimensiones a partir del encabezado.

- El formato **PGM (P2)** es un archivo de texto simple (ASCII) que almacena imágenes en tonos de gris.
- El archivo posee un **encabezado** estructurado de la siguiente forma:

1. **Versión:** El identificador `P2`.
2. **Comentarios:** Líneas opcionales que comienzan con `#` (deben ignorarse).
3. **Dimensiones:** Dos enteros que representan **Ancho** y **Alto**.
4. **Máximo:** Un entero que representa la intensidad máxima (generalmente 255).

- Después del encabezado, siguen los datos de los píxeles.

📌 **Importante**:

- **Lectura de Archivo:** Debes abrir el archivo indicado en el ejemplo utilizando la función `open()` de Python.
- **Orden de Salida:** A diferencia del orden presente en el archivo, la salida esperada debe estar en el formato de tupla: `(Altura, Ancho, Canales)`.
- Como los archivos PGM son en tonos de gris, el número de **Canales** es siempre **1**.
- Ver un simulador interactivo para esta cuestión en la **Figura 1.20** (ajusta las dimensiones para ver cómo se genera el encabezado ASCII).

1.17.10.1 🧠 Entendiendo el formato PGM

El formato PGM es uno de los más simples para el procesamiento de imágenes. Por ser en texto puro, permite visualizar los metadatos e incluso los valores de los píxeles abriendo el archivo en un bloc de notas:

Componente	Ejemplo	Significado
Número Mágico	`P2`	Identifica que es un PGM en formato de texto (ASCII).
Comentario	`# CREATOR...`	Línea informativa ignorada por el procesador.
Dimensiones	`397 343`	**397** columnas (Ancho) y **343** filas (Altura).
Intensidad	`255`	Define el valor del blanco puro (escala de 0 a 255).

1.17.10.2 📋 Tarea (especificación para VPL)

Entrada:

Ninguna entrada mediante teclado. El programa debe leer el archivo `"aula01fig03b.pgm"` presente en el directorio de ejecución.

Salida:

Una tupla que contenga `(Altura, Ancho, 1)`.

1.17.10.3 📌 Ejemplos

Nombre del Archivo	Salida Esperada	Observación
“aula01fig03b.pgm”	(343, 397, 1)	Nota la inversión del orden: Altura primero

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

Figura 1.20: Simulador EP01_10: Estructura de Archivo PGM (Encabezado ASCII P2 y Mapeo a Tupla Python)

> **Nota**
>
> El archivo necesario para este EP se descargará automáticamente del repositorio mediante el siguiente código:

```
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"  Archivo descargado: {f}")
        except urllib.error.HTTPError as e:
            print(f"  Error {e.code}: No encontrado en {url}")
```

```
%%writefile EP01_10.py
# tu solución
```

```
Overwriting EP01_10.py
```

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

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

1.17.11 EP01_11 ↗ Análisis de Vecindad: Filtro de Máximo 1D

En esta actividad, debes implementar un filtro morfológico simple de máximo que opera sobre una señal unidimensional (vector).

- Lee un entero **n**, que representa el tamaño del vector.
- Lee los **n** elementos enteros que componen el vector original **v1**.
- Crea un nuevo vector **v2**, donde cada posición i es el resultado de la comparación entre el elemento actual y sus vecinos inmediatos:

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

📌 **Importante**:

- **Bordes:** En los extremos del vector (índices 0 y $n-1$), la vecindad posee solo dos elementos (el propio y el único vecino disponible). En el índice 0, compara únicamente $v1[0]$ y $v1[1]$. En el último índice, compara solo $v1[n-2]$ y $v1[n-1]$.
- **Salida:** Imprime el encabezado "v2:" seguido de los valores del vector resultante, uno por línea.
- Consulta un simulador interactivo para esta pregunta en la **Figura 1.21** (pasa el mouse sobre los resultados para visualizar la ventana de vecindad utilizada en el cálculo).

1.17.11.1 🧠 ¿Por qué analizar vecinos?

En procesamiento de imágenes, el valor de un píxel rara vez es aislado; depende del contexto que lo rodea. El **Filtro de Máximo** es la base de la operación de **Dilatación** en morfología matemática, y sirve para:

Función	Efecto Visual
Realce	Expande estructuras brillantes y "engorda" objetos claros.
Eliminación de Ruido	Elimina pequeños puntos negros (ruido de "sal y pimienta" oscuro).
Relleno	Cierra pequeños agujeros o huecos en formas binarias.

1.17.11.2 📋 Tarea (especificación para VPL)

Entrada:

Un entero `n`.

En las líneas siguientes, los `n` elementos enteros del vector.

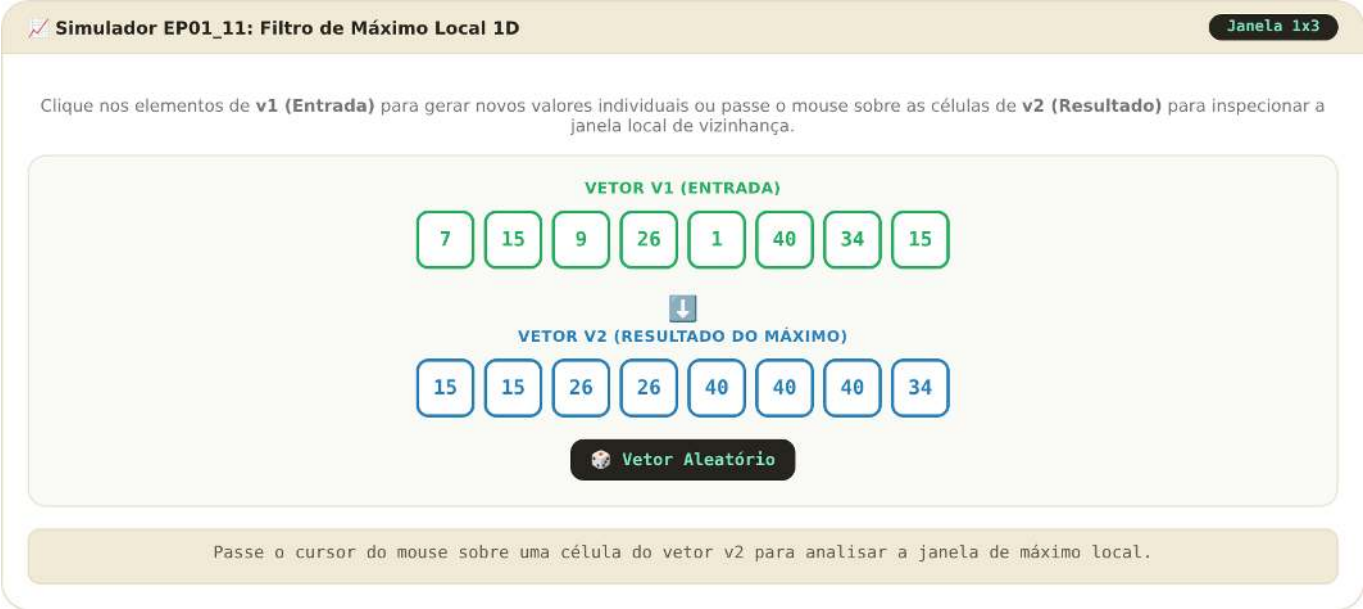
Salida:

La *cadena* `v2:` en la primera línea.

En las líneas siguientes, cada elemento de `v2` (uno por línea).

1.17.11.3 📌 Ejemplos

Entrada	Salida	Observación
5	v2:	En el índice 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.es/cap01/sim-ep0111-filtro-maximo.html

Figura 1.21: Simulador EP01_11: Filtro de Máximo Local 1D (Vecindad 1x3 con Condición de Borde)

```
%%writefile EP01_11.py
# tu solución
```

```
Overwriting EP01_11.py
```

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

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

Capítulo 2

De la Captura al Píxel - Muestreo, Cuantización y Conectividad

Executar Colab Abrir GitHub

Este capítulo profundiza en la comprensión de la imagen digital, transitando desde la naturaleza física de la captura hasta su representación matemática discreta. Investigamos cómo la luz se convierte en dato y cómo la organización espacial de los píxeles define las relaciones de vecindad y conectividad esenciales para algoritmos avanzados de Visión por Computador (VC).

2.1 Objetivos

Al final de este capítulo, podrá:

- Explicar el modelo físico de formación de la imagen basado en iluminación y reflectancia.
- Diferenciar los mecanismos de la visión humana y de los sensores digitales.
- Comprender los procesos de **muestreo** (discretización del espacio) y **cuantización** (discretización de la intensidad).
- Describir las relaciones topológicas entre píxeles: vecindad, adyacencia, conectividad y distancias.
- Realizar transformaciones geométricas básicas (traslación, rotación, escala) preservando la calidad.
- Visualizar en la práctica los efectos de la variación de la resolución espacial y de la profundidad de bits.

2.2 El Ojo y la Cámara - Elementos de la Percepción Visual

La formación de una imagen digital comienza con la captura de la luz reflejada por los objetos. Como se ilustra en Figura 2.1, tanto el ojo humano como las cámaras digitales siguen principios ópticos similares para enfocar la luz sobre una superficie sensible, aunque utilizan mecanismos biológicos y electrónicos distintos para la transducción de la señal.

2.2.1 Visión humana

El ojo funciona como un sistema óptico complejo: la luz atraviesa la córnea, el humor acuoso, la pupila (controlada por el iris) y el cristalino —que ajusta el enfoque dinámicamente— hasta llegar a la retina. En la retina se encuentran los fotorreceptores: los **conos** (6 millones), concentrados en la fóvea, son responsables de la visión de colores y detalles, mientras que los **bastones** (120 millones) garantizan la visión en condiciones de baja luminosidad (visión escotópica), detectando únicamente intensidades de gris. El punto ciego es la región de donde parte el nervio óptico, carente de receptores.

2.2.2 Sensores digitales

En las cámaras, los sensores de imagen desempeñan el papel de la retina. Los dos tipos más comunes son el **CCD** (*Charge-Coupled Device* - Dispositivo de Carga Acoplada) y el **CMOS** (*Complementary Metal-Oxide-Semiconductor* - Semiconductor de Óxido Metálico Complementario). El sensor está compuesto por una matriz de fotositios (píxeles) que acumulan carga eléctrica proporcional a la luz incidente.

Para la reconstrucción de colores, se utiliza el **Filtro de Bayer**, una matriz de filtros de color que permite que cada píxel capture únicamente un componente de color: rojo, verde o azul (RGGB - *Red, Green, Green, Blue*). Posteriormente, un **ADC** (*Analog-to-Digital Converter* - Convertidor Analógico-Digital) cuantifica esa carga en valores numéricos, definidos por una profundidad de bits (p. ej.: 8 bits, lo que resulta en 256 niveles de intensidad).

> **Curiosidad**
>
> Aunque el ojo humano tiene millones de receptores, la resolución de alta definición se restringe a la fóvea (visión central), equivalente a aproximadamente 120×120 píxeles. La percepción de una escena completa en alta resolución es el resultado de un intenso posprocesamiento realizado por el cerebro.

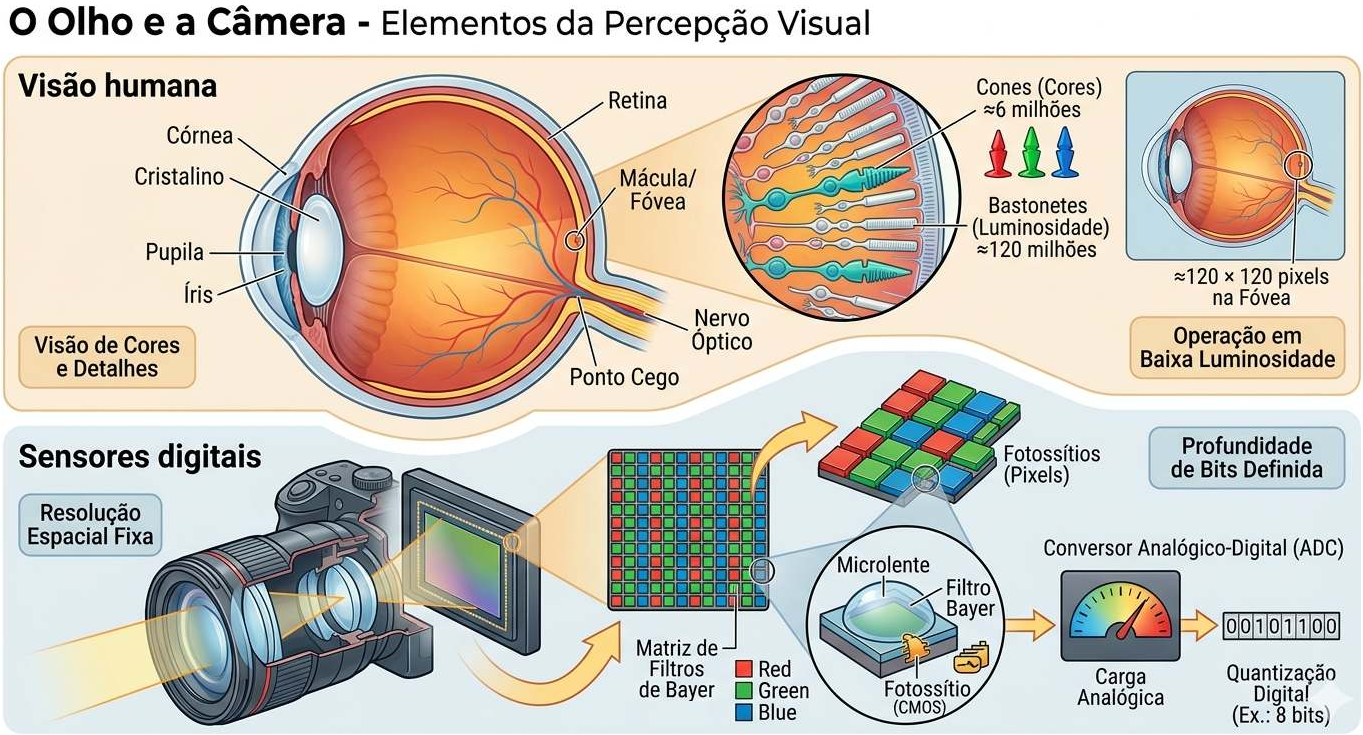


Figura 2.1: Comparativo didáctico entre el sistema visual biológico y el electrónico: en la parte superior, la anatomía del ojo humano destacando la retina y los fotorreceptores (conos y bastones); debajo, la estructura de una cámara digital detallando el sensor CMOS, la matriz de filtros de Bayer (RGGB) y el proceso de cuantización digital realizado por el ADC.

2.3 Ilusiones Ópticas: Los Desafíos de la Percepción Visual

Mientras que los sensores digitales capturan la intensidad de la luz de forma lineal y objetiva, el sistema visual humano interpreta la escena basándose en el contexto, las experiencias previas y los mecanismos biológicos de supervivencia. Las ilusiones ópticas no son "errores" del ojo, sino evidencias del intenso **posprocesamiento cerebral** realizado en la corteza visual.

2.3.1 Ambigüedad y Contexto

El cerebro busca constantemente dar sentido a patrones ambiguos. En el ejemplo del **Vaso de Rubin** (ver Figura 2.2), la percepción alterna entre la figura (el vaso) y el fondo (dos rostros), demostrando que no podemos procesar ambas interpretaciones simultáneamente. Por su parte, la ilusión del **Elefante de Shepard** juega con nuestra incapacidad de reconciliar líneas de contorno que sugieren volumen en posiciones lógicamente imposibles.

2.3.2 Brillo y Contraste Local

Muchas ilusiones derivan de la **inhibición lateral**, mecanismo mediante el cual las neuronas vecinas en la retina compiten entre sí para resaltar bordes. En la **Cuadrícula Scintillating**, puntos oscuros "fantasma" parecen aparecer en las intersecciones blancas debido a este procesamiento local de contraste.

La ilusión de la **Sombra del Tablero de Adelson** es quizás la más impactante para la VC: el cuadrado "A" y el cuadrado "B" poseen exactamente el mismo valor de gris en el sensor (o archivo digital), pero el cerebro "corrige" el brillo de "B" al comprender que se encuentra bajo una sombra proyectada, percibiéndolo como más claro.

2.3.3 Geometría y Perspectiva

La percepción de profundidad puede ser engañada por construcciones geométricas que desafían la lógica tridimensional desde un ángulo de visión específico. La **Escalera de Schröder** utiliza la ambigüedad de la perspectiva para crear un objeto que parece ascender o descender dependiendo de cómo se observe, evidenciando cómo nuestra interpretación de "arriba" y "abajo" depende del punto de fuga.

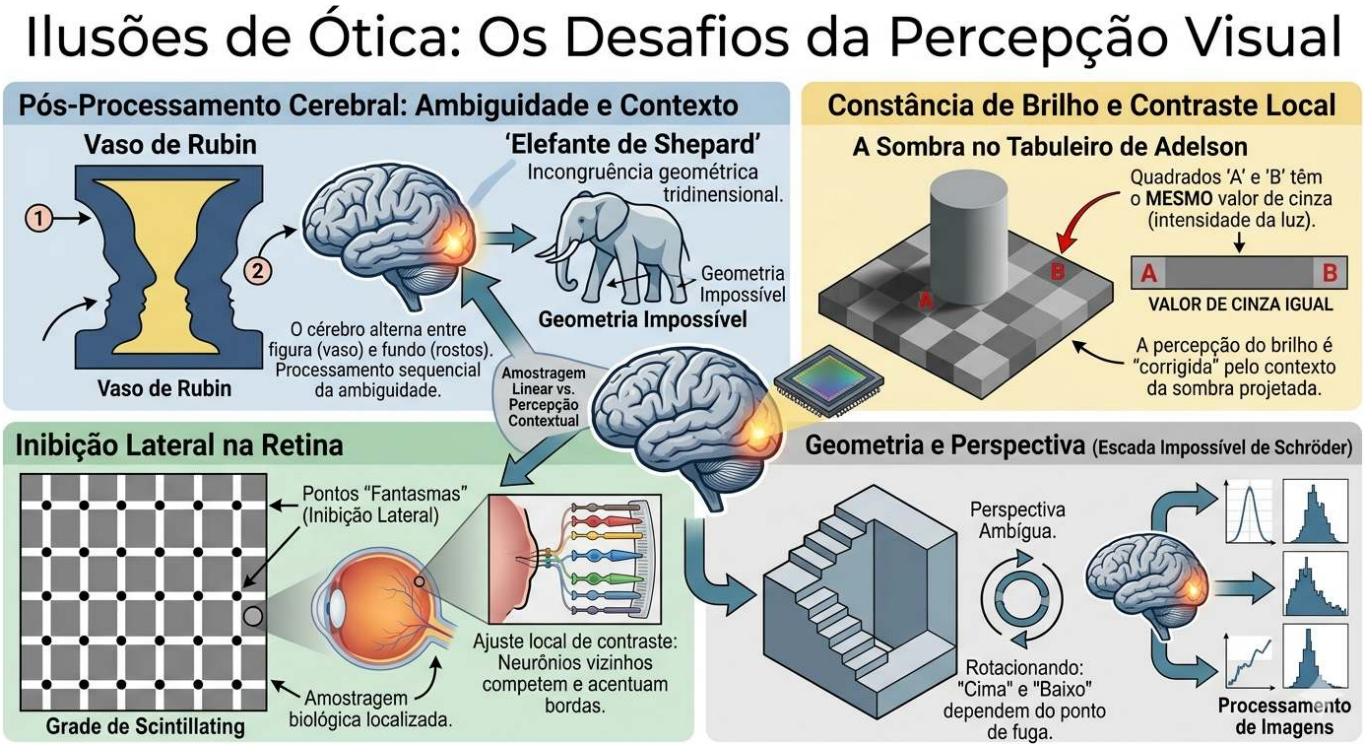


Figura 2.2: Colección de desafíos perceptivos: (superior izquierda) Vaso de Rubin — ambigüedad figura-fondo; (superior central) Elefante de Shepard — incongruencia geométrica; (superior derecha) Sombra de Adelson — constancia de brillo basada en el contexto; (inferior izquierda) Cuadrícula de puntos — inhibición lateral; (inferior derecha) Escalera de Schröder.

2.4 El Modelo Matemático de la Formación de la Imagen

Una imagen puede modelarse como el producto de dos funciones:

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

donde:

- $i(x, y)$ es la **iluminación** incidente sobre la escena (energía luminosa por unidad de área), determinada por la fuente de luz;

- $r(x, y)$ es la **reflectancia** del objeto (fracción de la luz reflejada), determinada por las propiedades ópticas de la superficie, con $0 < r(x, y) < 1$.

En la práctica, los dos componentes varían en escalas espaciales distintas: $i(x, y)$ tiende a variar lentamente en el espacio, mientras que $r(x, y)$ puede presentar variaciones abruptas asociadas a texturas, bordes y detalles finos. Las técnicas de PDI frecuentemente buscan separar o compensar estos componentes, como en métodos de corrección de iluminación no uniforme.

La Figura 2.3 ilustra cómo este modelo se manifiesta en imágenes digitales en color, representadas por múltiples canales espectrales (RGB) y, opcionalmente, por un canal adicional de transparencia (RGBA).

2.4.1 Representación Digital y Canales Espectrales

En imágenes digitales de color, la función $f(x, y)$ de la Ecuación 2.1 se representa mediante múltiples canales espectrales. En el estándar **RGB**, cada píxel almacena tres muestras independientes:

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

correspondientes a las intensidades de los componentes rojo (*Red*), verde (*Green*) y azul (*Blue*). En imágenes del tipo **RGBA**, se añade un cuarto canal:

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

donde $A(x, y)$ representa el canal **alfa** (*Alpha*), responsable de codificar la opacidad del píxel. Por convención, el valor $A = 0$ indica un píxel totalmente transparente, mientras que $A = 255$ (o 1) representa un píxel totalmente opaco.

Así, una imagen digital de color se estructura como una matriz multidimensional de dimensiones:

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

en la que cada posición (x, y) almacena las muestras asociadas a las propiedades ópticas de esa coordenada espacial.

La conversión entre rangos de intensidad es directa y se expresa como:

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

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

Convención de escala entre bibliotecas: Diferentes ecosistemas de programación adoptan rangos de valores y ordenamientos distintos para representar los canales digitales, tal como se sintetiza en la Tabla 2.1.

Tabla 2.1: Convenciones de escala y orden de canales en las principales bibliotecas de PDI.

Biblioteca	Tipo estándar	Rango de valores	Orden de las bandas	Forma del array/tensor
OpenCV	`uint8`	$[0, 255]$	**BGR**	$H \times W \times C$
Pillow	`uint8`	$[0, 255]$	RGB	$H \times W \times C$
NumPy	`uint8/float32`	$[0, 255]$ o $[0, 1]$	—	$H \times W \times C$
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	$H \times W \times C$

Descuidar estas discrepancias es una fuente frecuente de errores en los flujos de trabajo de VC — como, por ejemplo, inyectar una imagen leída mediante OpenCV (estándar BGR, tipo `uint8`) directamente en un modelo profundo de PyTorch que presupone el formato RGB normalizado.

2.4.2 Preparando el Entorno Práctico

El siguiente bloque carga el módulo `morph.py` del repositorio y demuestra, para un píxel sintético, las diferentes convenciones de escala y orden de canales adoptadas por las principales bibliotecas de PDI.

```
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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0

 Convenciones de escala por biblioteca
 NumPy / Pillow / OpenCV (uint8): [200 100  50] → [0, 255]
 scikit-image / PyTorch (float):  [0.78431373 0.39215686 0.19607843] → [0.0, 1.0]
 OpenCV: atención - lee en BGR:   [ 50 100 200] (canales invertidos)
```

2.4.3 Implementación de la Lectura de Imagen en `morph.py`

El siguiente fragmento muestra el código fuente de la función `mm.read`, permitiendo verificar directamente cómo la biblioteca `morph.py` implementa la lectura de imágenes.

```
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 y RGBA: Ejemplo Práctico con la Imagen Mandrill

El siguiente ejemplo carga la imagen Mandrill en formato RGB, construye artificialmente un canal alfa con gradiente horizontal y muestra ambas representaciones, ilustrando concretamente las estructuras matriciales $M \times N \times 3$ y $M \times N \times 4$.

```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"

#   Lectura RGB
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

#   Creación del canal alfa (gradiente horizontal)
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

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

#   Normalización float32 (estándar scikit-image / PyTorch)
img_rgb_norm = img_rgb.astype(np.float32) / 255.0
print(f"\nNormalizado (float32): rango=[{img_rgb_norm.min():.2f}, {img_rgb_norm.max():.2f}]")
print(f"Pixel (0,0) normalizado: {img_rgb_norm[0,0]}")

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

```
Normalizado (float32): rango=[0.05, 0.97]
Pixel (0,0) normalizado: [0.49019608 0.43137255 0.22745098]
```

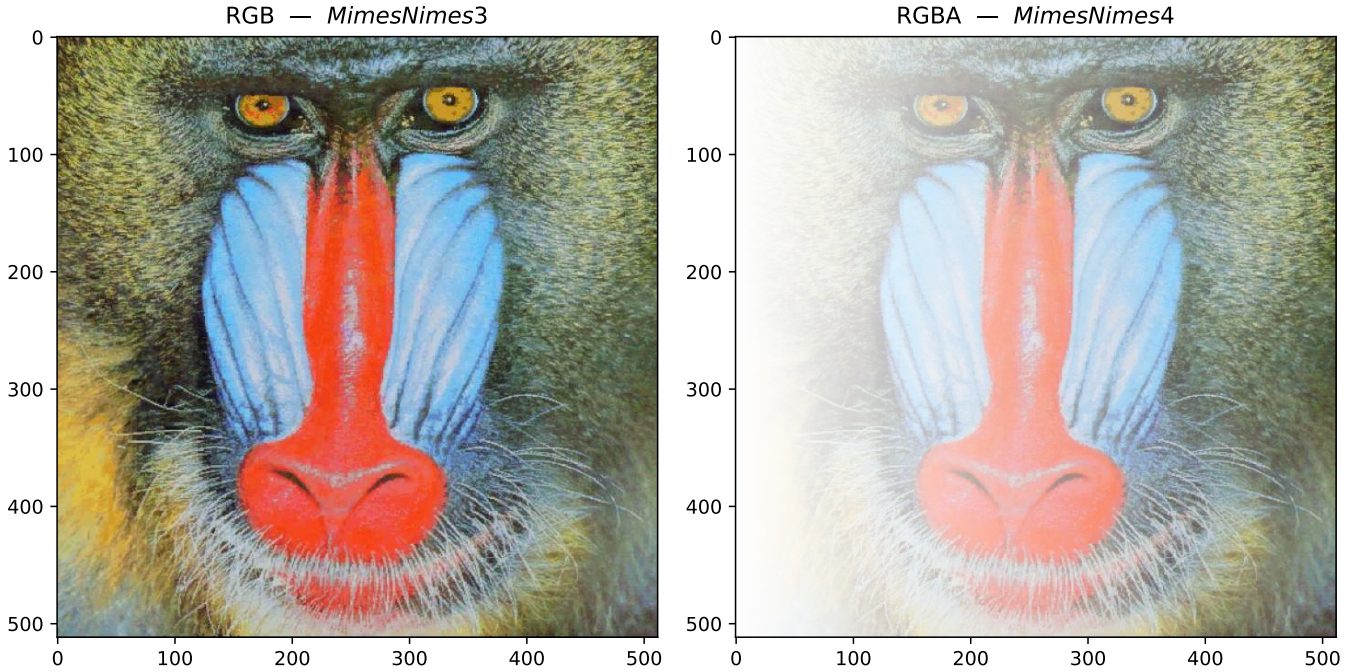


Figura 2.3: Imagen RGB (3 canales, $MimesNimes3$) y versión RGBA (4 canales, $MimesNimes4$) con transparencia alfa gradual de izquierda a derecha. Imagen Mandrill — USC SIPI Image Database (dominio público).

2.5 Digitalización: Muestreo y Cuantización

Para transformar una escena continua en una imagen digital, son necesarios dos procesos: muestreo y cuantización.

2.5.1 Muestreo - Discretización del espacio

El muestreo consiste en medir el valor de la función $f(x, y)$ en puntos igualmente espaciados, formando una matriz de M filas (altura) y N columnas (ancho). Cada elemento de esta matriz es un **píxel**. La **resolución espacial** está dada por $M \times N$. Cuanto mayor es la resolución, más detalles espaciales se preservan, pero también mayor es el costo computacional y de almacenamiento.

2.5.2 Cuantización - Discretización de la intensidad

La cuantización asocia a cada píxel un valor numérico discreto, generalmente representado por un entero de b bits. La **profundidad de bits** define el número de niveles de intensidad: 2^b. Las imágenes en tonos de gris suelen usar 8 bits (256 niveles). Las imágenes en color usan tres canales de 8 bits (24 bits en total).

Ilustración: Si usamos solo 1 bit por píxel (blanco y negro), perdemos todos los tonos intermedios. Con 2 bits (4 niveles) ya se perciben degradados gruesos. Con 8 bits, el ojo humano difícilmente percibe la discretización (visión continua).

⚠ Error de cuantización

Error de cuantización es la diferencia entre el valor analógico real y el valor discreto asignado. Se manifiesta como ruido de cuantización, visible en regiones con gradiente suave cuando se usan pocos bits.

2.5.3 Efectos del muestreo y la cuantización

Los siguientes experimentos muestran cómo la reducción de la resolución espacial (submuestreo) y de la profundidad de bits degradan la calidad visual. Utilice el código para explorar diferentes factores y niveles de gris.

```
# Imagen de ejemplo (barbudo-rayado) - base de las figuras de muestreo,
# cuantización y transformaciones geométricas de este capítulo.
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)  # recorte p/ ver detalles
print(f"Imagen original: {img_color.shape}")
```

```
Imagen original: (3067, 2047, 3)
```

```
import numpy as np

# Relectura como objeto PIL para acceso a los metadatos EXIF (solo Python).
img_pil   = mm.read(caminho, pil=True)   # retorna PIL Image con EXIF
img_numpy = np.array(img_pil)
exif      = img_pil._getexif()           # ahora funciona - img_pil es PIL Image
print(f"Tipo de la imagen: {type(img_color)}")
```

```
Tipo de la imagen: <class 'numpy.ndarray'>
```

El experimento presentado en la Figura 2.4 ilustra el compromiso entre la **resolución espacial** y el **costo de almacenamiento** en memoria. El código utiliza la técnica de submuestreo por rebanado (*slicing*) para reducir la matriz original de píxeles según un factor f, lo que resulta en un ahorro de memoria — por ejemplo, un factor $f = 8$ reduce el tamaño del dato en 64 veces (8^2). Para fines de comparación visual, las imágenes reducidas se restauran a las dimensiones originales (512×512) mediante la interpolación por **vecino más cercano** (`nearest`). Este proceso no recupera la información perdida, pero hace evidente el efecto de ***aliasing*** y la estructura de bloques (pixelización) generada por la baja densidad de datos de la matriz muestreada.

```
def subsample_simple(image, f):
    # Submuestreo mediante segmentación (slicing)
    reduced = mm.subsample(image, f)

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

    # Restablece el tamaño para visualización (H, W originales)
    res = mm.resize(reduced, (image.shape[1], image.shape[0]), method='nearest')
    return res, label
```

```
factors = [1, 4, 8, 12]

# Genera los resultados y los separa en listas para mm.show
results = [subsample_simple(img_gray, f) for f in factors]
imgs_list = [r[0] for r in results]
titles_list = [r[1] for r in results]

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

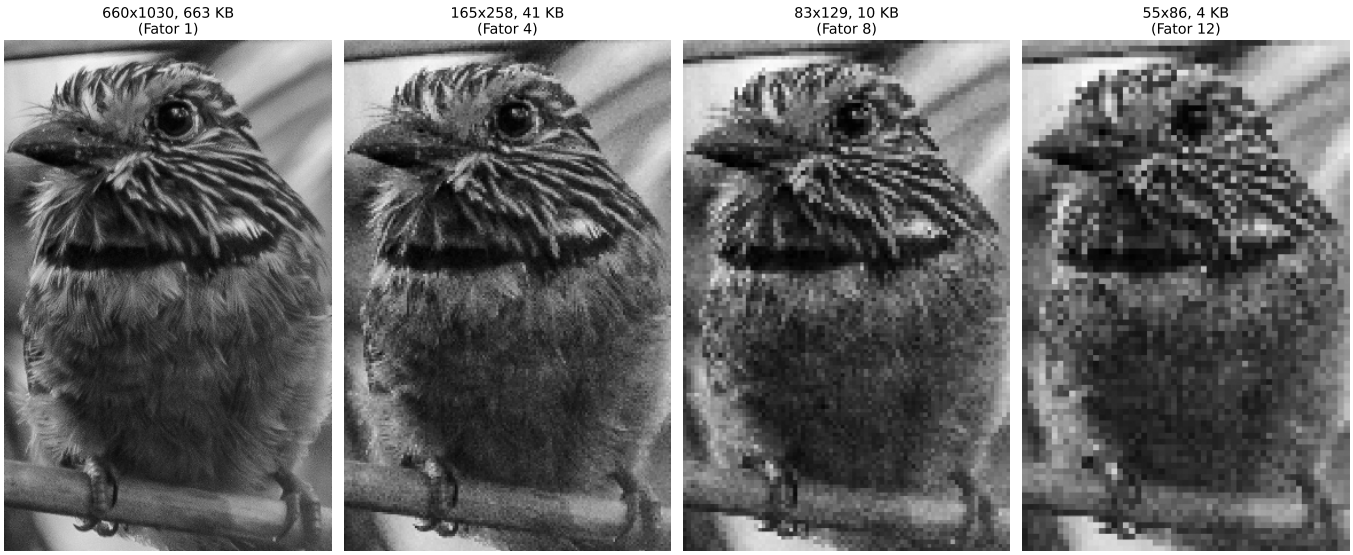


Figura 2.4: Efecto del submuestreo. Los títulos muestran las dimensiones (W x H) y el tamaño de la matriz en memoria (KB).

El experimento en la Figura 2.5 se centra en la **cuantización de intensidad**, el proceso de discretización de la amplitud de la función $f(x, y)$. Mientras que el submuestreo afecta la rejilla espacial, la reducción de la profundidad de bits limita la cantidad de tonos de gris disponibles para representar el brillo.

Al reducir la profundidad de 8 bits (256 niveles) a valores menores, surge el efecto de **posterización**, donde los gradientes suaves de una escena se sustituyen por transiciones abruptas. En el límite de 1 bit, la imagen se vuelve estrictamente binaria, preservando únicamente la silueta y perdiendo detalles de textura y volumen.

```
def quantize_simple(image, bits):
    """Reduce la profundidad de bits y calcula metadatos de memoria."""
    levels = 2 ** bits
    # Normalización y cuantización uniforme
    quantized = (np.floor(image / 256 * levels) / levels * 255).astype(np.uint8)

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

    return quantized, label

# Lista de bits para prueba (8 es el estándar, 1 es el binario)
bits_test = [8, 4, 2, 1]

# Genera los resultados y los separa en listas para el mm.show
results_q = [quantize_simple(img_gray, b) for b in bits_test]
imgs_q = [r[0] for r in results_q]
titles_q = [r[1] for r in results_q]
```

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

Figura 2.5: Efecto de la reducción de la profundidad de bits. Los títulos muestran la cantidad de bits, niveles y el tamaño en memoria (KB).

2.5.4 Análisis Técnico

- **Dominio vs. Codominio**: Nótese que la resolución espacial (dimensiones de la matriz) permanece constante en 512x512; lo que se altera es únicamente el codominio de la función de la imagen.
- **Constancia de Memoria**: Obsérvese en los títulos que el tamaño en KB no disminuye. Esto ocurre porque NumPy almacena cada píxel cuantizado en un contenedor de 8 bits (`uint8`), independientemente de que el valor real sea solo 0 o 1.
- **Percepción**: La degradación visual se vuelve crítica por debajo de 4 bits, donde el ojo humano comienza a percibir las "fronteras" artificiales creadas por la falta de tonos intermedios.

La limitación de tipos de datos menores que un byte en el ecosistema Python/NumPy se debe a la arquitectura del hardware, que direcciona la memoria en bloques de 8 bits (Bytes). Para mantener la compatibilidad con OpenCV y garantizar la eficiencia, incluso los elementos binarios se mapean a contenedores de 1 byte (`uint8` o `bool8`).

Aunque lenguajes como ANSI C permiten la compactación de 8 píxeles por byte (*bit-packing*), este enfoque exige la descompactación constante para los cálculos e impone una alta complejidad en la manipulación de punteros. Según la Tabla 2.2, se privilegia el uso de `uint8` por la facilidad en el acceso a vecinos y por la versatilidad en transformaciones geométricas. Además, los métodos nativos de NumPy y OpenCV ejecutan el procesamiento internamente a bajo nivel (C/C++), lo que hace que las operaciones vectorizadas sean más rápidas que las implementaciones manuales con bucles anidados en Python.

Tabla 2.2: Comparativo entre estrategias de compactación y eficiencia de procesamiento.

Característica	Python (NumPy/OpenCV)	ANSI C (Bit-packing)
Unidad más pequeña	1 Byte (8 bits)	1 Bit
Memoria (Binaria)	256 KB (para 512x512)	32 KB (para 512x512)
Velocidad	Alta (Vectorización en C)	Variable (Lenta si hay bit-shift)
Complejidad	Baja: Métodos listos	Alta: Punteros y Máscaras

2.6 Relaciones entre Píxeles - Topología de la Imagen

Los píxeles no son elementos aislados; sus posiciones relativas definen conceptos importantes para el procesamiento.

2.6.1 Vecindad

Dado un píxel de coordenadas (x, y), se definen dos tipos principales de vecindad (para imágenes en *grid* rectangular):

- **Vecindad-4** (von Neumann): incluye los píxeles en las posiciones $(x-1, y)$, $(x+1, y)$, $(x, y-1)$, $(x, y+1)$.
- **Vecindad-8** (Moore): incluye los ocho píxeles adyacentes (añade los cuatro diagonales).

La elección de la vecindad influye en operaciones como detección de bordes, cálculo de gradientes y conectividad.

```
# # Creación de una matriz 3x3 para ejemplo topológico
# viz = np.zeros((3, 3), dtype='uint8')

# # Definiendo Vecindad-4 (N4) con valor diferente para resaltado
# viz[0, 1] = viz[2, 1] = viz[1, 0] = viz[1, 2] = viz[1, 1] = 255

# o simplemente (prueba con números como argumentos):
viz = mm.secross()

# Visualización de la matriz para análisis de coordenadas
mm.drawImgPlt(viz, scale=40)
```

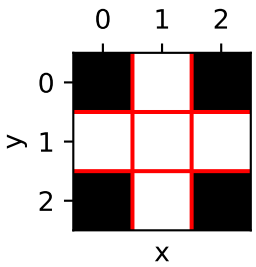


Figura 2.6: Ilustración de vecindades 4 en una matriz 3x3. En el centro (1,1), el píxel de interés.

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

2.6.2 Adyacencia, conectividad y caminos

Dos píxeles son **adyacentes** si están en contacto según una vecindad definida **y** satisfacen un criterio de valor (ej.: mismo nivel de intensidad). Una **conectividad** define una relación de equivalencia entre píxeles que forman una región conexa. Un **camino** es una secuencia de píxeles adyacentes.

La **conectividad-4** (N4) y la **conectividad-8** (N8) pueden producir resultados diferentes en la segmentación y en el cálculo de componentes conexas (etiquetado). Por ejemplo, un patrón de tablero de ajedrez puede ser completamente disconexo en N4, pero totalmente conexo en N8.

2.6.3 Distancias entre píxeles

Las métricas de distancia son fundamentales para cuantificar la proximidad física y la conectividad entre los elementos que componen la rejilla digital. Como se demuestra en la Tabla 2.3, la elección de la métrica define el coste de desplazamiento entre píxeles y altera el comportamiento de los algoritmos de segmentación y análisis morfológico.

Para medir la distancia entre dos píxeles $p(x_1, y_1)$ y $q(x_2, y_2)$, se utilizan diferentes funciones métricas que imponen restricciones de movimiento distintas sobre la *cuadrícula*:

Tabla 2.3: Comparativa de métricas de distancia aplicadas a la malla de píxeles.

Métrica	Definición	Interpretación
Euclidiana	$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$	Distancia **exacta** en línea recta (continua)
Manhattan (*City block*)	$\lvert x_1 - x_2\rvert + \lvert y_1 - y_2\rvert$	Movimientos horizontales + verticales
Chebyshev (*Tablero*)	$\max(\lvert x_1 - x_2\rvert, \lvert y_1 - y_2\rvert)$	Mayor desplazamiento entre los ejes

Estas distancias se aplican en diversos contextos de PDI, incluyendo algoritmos de interpolación geométrica, transformadas de distancia, crecimiento de regiones y análisis de formas.

ℹ Ejemplo práctico

Considerando dos píxeles con desplazamientos relativos $\Delta x = 3$ y $\Delta y = 4$:

- **Euclidiana**: $\sqrt{3^2 + 4^2} = 5$ (hipotenusa del triángulo rectángulo).
- ***Manhattan***: $3 + 4 = 7$ (suma de los catetos).
- ***Chebyshev***: $\max(3, 4) = 4$ (predominio del mayor desplazamiento).

2.7 Almacenamiento de Imágenes

La elección del formato de archivo es un paso decisivo en el flujo de procesamiento, ya que determina cómo se preservarán o descartarán los datos de muestreo y cuantización. Como se presenta en la Tabla 2.4, cada extensión equilibra de forma distinta la fidelidad de los datos y la eficiencia de almacenamiento.

Tabla 2.4: Principales formatos de almacenamiento de imágenes digitales y sus aplicaciones en PDI.

Formato	Características	Uso típico
PGM	Formato simple de mapa de grises (texto o binario).	Investigación académica y herramientas Unix.
BMP	Sin compresión (o compresión simple).	Windows, aplicaciones heredadas.
PNG	Compresión **sin pérdidas** (*lossless*).	Web, imágenes con transparencia.
JPEG	Compresión **con pérdidas** (*lossy*), ideal para fotografías.	Fotos, cámaras digitales.
TIFF	Admite múltiples capas y compresión variada.	Edición, archivado.
RAW	Datos brutos del sensor, sin procesamiento.	Fotografía profesional.
DICOM	Estándar médico con metadatos clínicos integrados (paciente, equipo, protocolo).	Radiología, tomografía, resonancia magnética.

Los metadatos de una imagen incluyen parámetros como ancho, alto, profundidad de bits y codificación de color. En formatos científicos, también se preservan información de calibración y detalles de la captura. Al utilizar la función `mm.read()`, la biblioteca `morph` preserva automáticamente estos datos para que se respeten las propiedades originales de la imagen.

En contextos científicos y hospitalarios, se privilegia el estándar **DICOM** (*Digital Imaging and Communications in Medicine*) para garantizar que no haya pérdida de precisión diagnóstica. Repositorios públicos como **The Cancer Imaging Archive (TCIA)**, la **Alzheimer's Disease Neuroimaging Initiative (ADNI)** y **PhysioNet** ponen a disposición vastos conjuntos de datos en este formato, incluidos metadatos clínicos anonimizados que son esenciales para la investigación científica.

2.7.1 Ejemplo: Extracción de Metadatos y Localización GPS

A diferencia de la matriz de píxeles pura obtenida mediante la lectura convencional —como en la imagen del pájaro presentada al inicio de este capítulo—, el uso del argumento `pil=True` en el método `mm.read()` altera la naturaleza del objeto devuelto (ver Figura 2.7). Mientras que el estándar (`pil=False`) devuelve un `numpy.ndarray` RGB, la lectura con `pil=True` devuelve un objeto especializado de la biblioteca Pillow, capaz de interpretar el encabezado **EXIF**.

El encabezado **EXIF** (*Exchangeable Image File Format*) funciona como un repositorio técnico de la captura, permitiendo que el objeto Pillow interprete una amplia gama de información que va mucho más allá de las coordenadas GPS. Al utilizar `pil=True`, el sistema obtiene acceso al "ADN" de la imagen, incluyendo metadatos de hardware (marca y modelo de la cámara), configuraciones ópticas (apertura, distancia focal y tiempo de exposición) y parámetros de iluminación (uso de flash y balance de blancos).

Esta distinción es importante en el PDI, ya que transforma la matriz de muestreo en un conjunto de datos contextualizado, donde las características físicas del sensor y del lente pueden utilizarse para normalizar brillos o corregir distorsiones geométricas.

```
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. Lectura - preserva objeto PIL con 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)      # guarda preservando EXIF
else:
    img_obj = mm.read(caminho, pil=True)

img_numpy = np.array(img_obj)      # conversión a NumPy

# 2. Extracción y conversión de GPS (Etiqueta 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}")
```

```python
# 3. Diagnóstico de tipos, dimensiones y acceso a píxeles
print(f"\nTipo PIL  : {type(img_obj)}   | Dimensiones (x,y): {img_obj.size}")
print(f"Pillow (0,0): {img_obj.getpixel((0, 0))}")
print(f"Tipo NumPy: {type(img_numpy)} | Dimensiones [y,x,c]: {img_numpy.shape}")
print(f"NumPy [0,0]: {img_numpy[0, 0]}")

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

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

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

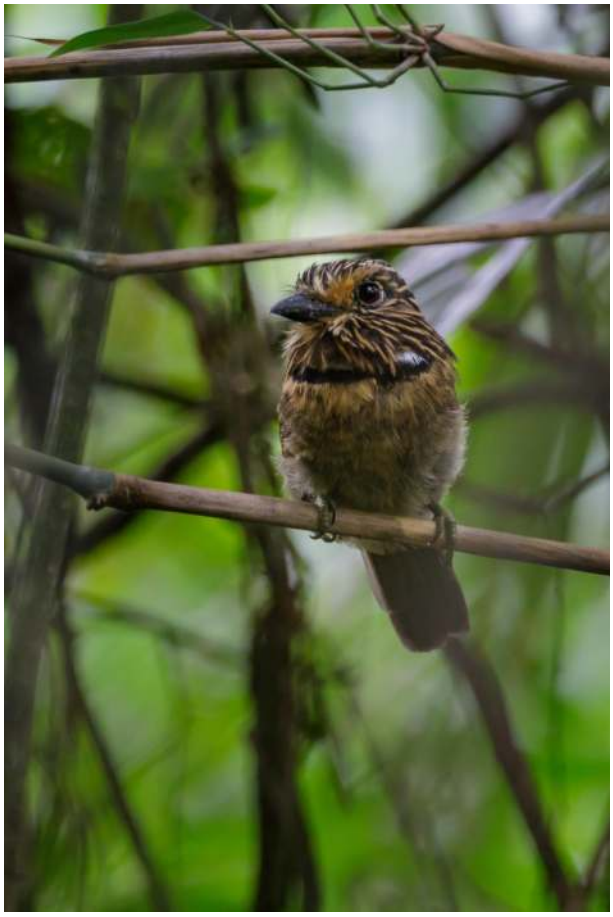


Figura 2.7: Área de Protección Ambiental Quilombos do Médio Ribeira - Barbudo-rajado (Malacoptila striata). Crédito: Thomas Fuhrmann (CC BY-SA 4.0).

Nota Pedagógica: La sutil diferencia de las dimensiones

Observe que la representación de las dimensiones cambia según la estructura de datos utilizada:

- **En Pillow (`.size`):** Devuelve `(Ancho, Alto)` — en el ejemplo: `(2047, 3067)`. Es una visión orientada al archivo de imagen.
- **En NumPy (`.shape`):** Sigue la convención matemática de matrices: `(Filas/Alto,`

`Columnas/Ancho, Canales)` — en el ejemplo: `(3067, 2047, 3)`.
Esta distinción es fundamental para evitar errores de indexación al implementar filtros manuales. Mientras el objeto Pillow carga el **"dónde"** y el **"cuándo"** (contexto), el array NumPy carga el **"cuánto"** de luz (intensidad) que existe en cada punto de la imagen.

2.7.2 ¿Por qué es importante esta separación?

Al cargar una imagen por el camino convencional (`pil=False`), el resultado es un `numpy.ndarray`, que contiene estrictamente los valores numéricos resultantes de la **cuantización** y **muestreo**. Sin embargo, al utilizar `pil=True`, el `mm.read()` devuelve un objeto de la clase `PIL.JpegImagePlugin.JpegImageFile`.

Esta clase mantiene el archivo "abierto" para permitir el acceso al contexto de la captura antes de que los datos se conviertan en una matriz bruta. Note que el píxel `(0, 0)` es idéntico en las dos representaciones — `(58, 96, 0)` en Pillow y `[58, 96, 0]` en NumPy —, confirmando que ambos describen los mismos datos, solo con interfaces distintas. Esta separación es vital: los píxeles sirven para algoritmos; los metadatos sirven para georreferenciación, catalogación científica y correcciones basadas en el hardware de adquisición.

Para inspeccionar todos los metadatos EXIF de un objeto Pillow:

```
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 Transformaciones Geométricas Básicas

Las transformaciones geométricas alteran la posición de los píxeles, manteniendo los valores de intensidad. Son fundamentales para la alineación, la corrección de distorsiones y el aumento de datos (*data augmentation*) en el aprendizaje automático.

Una **transformación afín** es cualquier mapeo que preserve la colinealidad (puntos sobre una recta permanecen sobre una recta) y las razones de distancias entre puntos colineales. En 2D, toda transformación afín puede expresarse en **coordenadas homogéneas** mediante una matriz 3×3:

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

La submatriz 2×2 superior izquierda $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ controla la rotación, la escala y el cizallamiento; el vector $(t_x, t_y)^\top$ controla la traslación. Las transformaciones geométricas más utilizadas en PDI —traslación, rotación y escala— son casos particulares de $\mathbf{T}$, y pueden **componerse** mediante la multiplicación de matrices, en el orden $\mathbf{T} = \mathbf{T}_n \cdots \mathbf{T}_2 \mathbf{T}_1$.

ℹ Transformación inversa e interpolación

En la implementación práctica (`cv2.warpAffine`), se aplica la **transformación inversa**: para cada píxel (x', y') de la imagen destino, se calcula la posición de origen $(x, y) = \mathbf{T}^{-1}(x', y')$ y se interpola el valor. Esto evita huecos en la imagen resultante causados por el mapeo directo de píxeles enteros a posiciones no enteras.

2.8.1 Traslación

La traslación es la transformación afín más simple: desplaza todos los píxeles mediante un vector (t_x, t_y). En coordenadas homogéneas, se expresa mediante la matriz:

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

La tercera fila de la matriz garantiza que la operación permanezca en el espacio afín, permitiendo que la traslación, la rotación y la escala se combinen mediante una simple multiplicación de matrices. En la práctica, `cv2.warpAffine` utiliza solo las dos primeras filas (matriz 2×3), ya que la tercera es siempre $[0, 0, 1]$.

Los píxeles desplazados más allá del área original se descartan; las áreas descubiertas se rellenan con 0 (negro). Ver un ejemplo en la Figura 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", "Traslación (50,50)", "Traslación (100,50)"],
        cols=3, figsize=(16, 12))
```

Figura 2.8: Ejemplo de traslación de la imagen del pájaro con desplazamientos (50,50) y (100,50).

2.8.2 Rotación

La rotación por un ángulo θ alrededor de un punto central (c_x, c_y) se compone de tres transformaciones afines: traslación al origen, rotación pura y traslación de regreso. La matriz resultante es:

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

En la implementación, `cv2.getRotationMatrix2D` genera directamente las dos primeras filas de $\mathbf{T}_{\text{rot}}$ (matriz 2×3 para `warpAffine`), aceptando también un factor de escala s que multiplica $\cos\theta$ y $\sin\theta$. Ver ejemplo en la Figura 2.9.

Dado que la rotación desplaza píxeles a nuevas posiciones no enteras, `cv2.warpAffine` necesita estimar el color de cada píxel de destino a partir de los vecinos — proceso llamado **interpolación**. El parámetro `interp` controla este comportamiento:

- **`nearest`** (`INTER_NEAREST`): asigna el valor del píxel más cercano. Rápido, pero produce bordes dentados (*aliasing*) en bordes diagonales.
- **`bilinear`** (`INTER_LINEAR`, predeterminado): promedio ponderado de los 4 vecinos más cercanos. Equilibra calidad y rendimiento — adecuado para la mayoría de los casos.
- **`bicubic`** (`INTER_CUBIC`): considera los 16 vecinos en una superficie cúbica. Produce bordes más suaves, al costo de mayor procesamiento.

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

Figura 2.9: Ejemplo de rotación de la imagen del pájaro en 30° y 45° usando interpolación bilineal.

2.8.3 Escala (Redimensionamiento)

El redimensionamiento por factores (s_x, s_y) es una transformación afín con matriz:

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

Cuando $s > 1$ (ampliación), los píxeles de la imagen destino se mapean a posiciones no enteras en el origen — lo que exige interpolación para estimar el valor. Cuando $s < 1$ (reducción), múltiples píxeles de origen

contribuyen a un único píxel destino — lo que exige **decimación**. Los mismos tres métodos descritos en la rotación están disponibles en `mm.resize`, con la diferencia de que aquí el impacto visual es más perceptible: en la ampliación, `nearest` produce efecto de bloques (*pixelation*), mientras que `bicubic` preserva mejor la nitidez de los bordes, conforme a la Tabla 2.5:

Tabla 2.5: Métodos de interpolación disponibles en `mm.resize` y sus respectivos números de vecinos utilizados en el cálculo.

Método	Vecinos utilizados	Característica
`'nearest'`	1	Rápido; produce efecto de bloques (*pixelation*)
`'bilinear'`	4	Buen compromiso calidad/costo; bordes suaves
`'bicubic'`	16	Mayor nitidez; preferido en software profesional

El parámetro `size_or_factor` acepta tanto un escalar (factor uniforme, p. ej., `0.5` para reducir a la mitad) como una tupla `(ancho, alto)` para dimensiones absolutas.

```
# 1. Recorte de la región del pico
y, x, offset = 210, 40, 40
crop = img_gray[y-offset:y+offset, x-offset:x+offset]
print(f"Imagen: {img_gray.shape} | Crop: {crop.shape}")

# 2. Ampliar 4× con 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. Exhibición comparativa
mm.show(
    [crop, crop_nearest, crop_bilinear],
    titles=["Original (recorte)", "Vecino más cercano (4×)", "Bilineal (4×)"],
    cols=3, figsize=(16, 12), dpi=200
)
```

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

Figura 2.10: Comparación de interpolación con zoom en el detalle del ojo (recorte 60×60, ampliado 4×). Observe el efecto de bloques en el vecino más cercano vs. la suavización en la bilineal.

2.8.4 Cisallamiento (*Shear*)

El cisallamiento es una transformación afín que distorsiona la imagen al desplazar cada píxel proporcionalmente a su posición en un eje. La matriz general combina el cisallamiento horizontal (sh_x) y vertical (sh_y):

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

Para $sh_x \neq 0$ y $sh_y = 0$, cada fila se desplaza horizontalmente de manera proporcional a su posición vertical, produciendo el efecto de "inclinación" característico. Ver ejemplo en la Figura 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)", "Combinado (0.2, 0.2)"],
    cols=4, figsize=(16, 12)
)
```

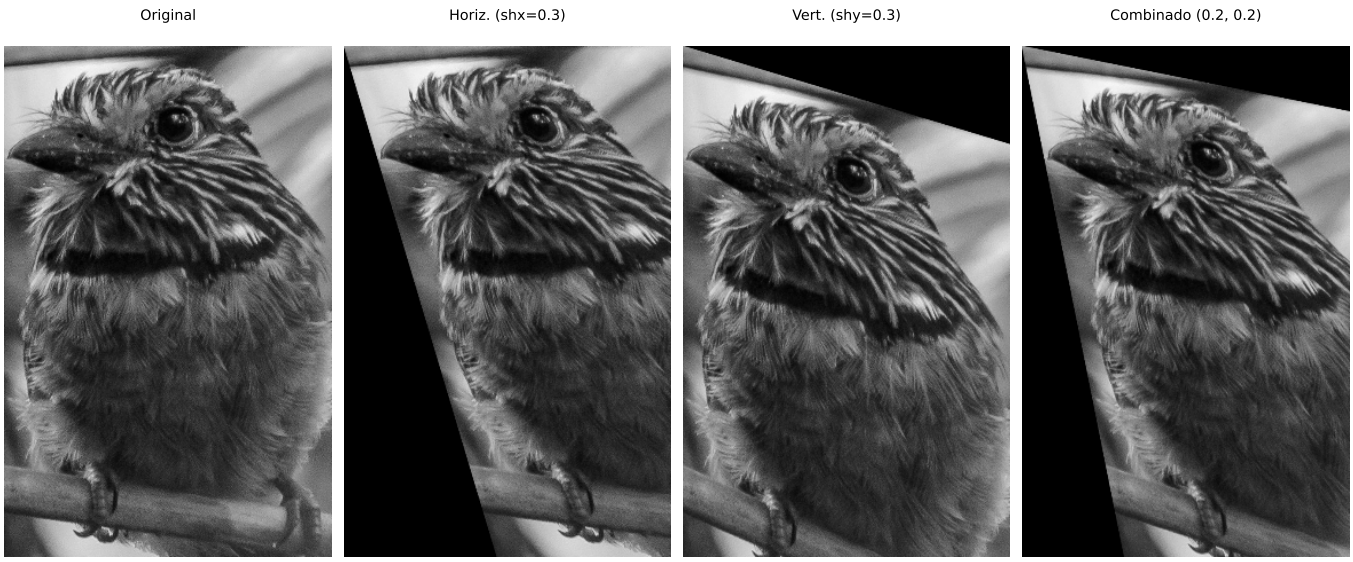


Figura 2.11: Ejemplo de cizallamiento de la imagen del pájaro: horizontal (shx=0.3), vertical (shy=0.3) y combinado (shx=0.2, shy=0.2).

2.9 Resumen

En este capítulo se presentaron los fundamentos de digitalización y topología de imágenes:

- **Formación de la imagen:** $f(x, y) = i(x, y) \cdot r(x, y)$.
- **Muestreo:** discretización del espacio $\rightarrow$ resolución espacial $M \times N$.
- **Cuantización:** discretización de la intensidad $\rightarrow$ profundidad de bits b.
- **Relaciones topológicas:** vecindad-4, vecindad-8, conectividad, distancias (Euclidiana, Manhattan, Chebyshev).
- **Transformaciones geométricas:** traslación, rotación, escala (con interpolación bilineal o vecino más próximo).
- **Formatos de archivo:** BMP, PNG, JPEG, TIFF, RAW; cada uno con diferentes compromisos entre calidad y tamaño.

El Capítulo 3 abordará **operaciones espaciales** como convolución, filtrado y morfología matemática (erosión, dilatación).

2.10 🤖 Uso del Gemini Notebook como Tutor Complementario

En esta edición, incentivamos el uso del **Gemini Notebook** como herramienta complementaria de aprendizaje. Esta herramienta de IA utiliza exclusivamente los documentos proporcionados por el autor como base de conocimiento, garantizando respuestas coherentes con el contenido del libro.

Para cada capítulo, hemos preparado un proyecto específico en la plataforma. Para una experiencia de estudio ampliada, utilice el siguiente acceso:

🎓 Estudie con el Tutor Inteligente

Para interactuar con el contenido de este capítulo, acceda al siguiente enlace. El entorno contiene materiales didácticos en diferentes formatos, generados a partir del **PDF** del capítulo. En la plataforma, explore especialmente las opciones **Guía de Estudio** y **Conversación** para profundizar su comprensión.
🚀 ACCEDER A GEMINI NOTEBOOK: CAPÍTULO 02

🌐 **Idioma y Lenguaje de Programación**

El proyecto de este capítulo en el Gemini Notebook fue construido únicamente con el texto en **portugués** y los ejemplos de código en **Python**. Si está estudiando la edición en inglés o francés, o siguiendo la ruta en C++, las respuestas del tutor pueden no corresponder exactamente a la versión que está leyendo.

⚠️ **Aviso sobre Contenido Generado por IA**

La IA es una poderosa aliada en los estudios, pero el contenido generado puede contener **errores o imprecisiones**. Consulte siempre **libros, artículos científicos y otras fuentes académicas confiables** para validar la información. Siempre que sea posible, ejecute los ejemplos prácticos proporcionados en este capítulo para verificar los resultados.

2.11 Lista de Ejercicios

1. **(15%)** Explique, con sus propias palabras, la diferencia entre **muestreo** y **cuantización**. Dé un ejemplo concreto de cada uno en el contexto de una imagen digital.
2. **(15%)** Considere una imagen con resolución espacial de 1024×768 píxeles y profundidad de 24 bits (8 bits por canal RGB). Calcule el tamaño total no comprimido de la imagen en bytes y en megabytes.
3. **(20%)** Utilizando el código del laboratorio, modifique el factor de submuestreo a 3 y a 6. Describa visualmente qué ocurre con los bordes de los objetos. ¿Qué es el efecto de *aliasing*?
4. **(20%)** Para la imagen en escala de grises, aplique cuantización con 3 bits (8 niveles) y 5 bits (32 niveles). Compare los resultados y explique por qué 5 bits ya puede considerarse suficiente para muchas aplicaciones.
5. **(15%)** Dados dos píxeles $A = (10, 20)$ y $B = (15, 25)$, calcule las distancias Euclidiana, Manhattan y Chebyshev entre ellos.
6. **(15%)** Usando la función `mm.rotate`, gire la imagen del pájaro en ángulos de 90°, 180° y 270° con interpolación bilineal. Compare con la rotación usando `method='nearest'`. ¿En qué situaciones la interpolación de vecino más cercano sigue siendo útil?

Referencias del Capítulo

La fundamentación teórica de este capítulo se basa en las siguientes obras:

- Gonzalez (2018) para los conceptos de muestreo, cuantización y relaciones entre píxeles.
- Szeliski (2022) para transformaciones geométricas y conectividad.

- Bradski (2008) para la implementación práctica con OpenCV y `morph.py`.

[req 928345] Traduce este Markdown de Portugués a Español:

Executar Colab | Abrir GitHub

2.12 💻 Parte Práctica con Ejercicios de Programación

🎯 Objetivo de este Cuaderno

El cuaderno permite desarrollar, validar, organizar y probar soluciones de **Ejercicios de Programación (EPs)** en entornos interactivos, como Colab, con los mismos casos de prueba de Moodle, copiando allí únicamente al momento de registrar la nota oficial.

Download

Descargue `morph.py` y `testsuite.py` ejecutando la celda a continuación:

```
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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Ejecutando las pruebas

Para evaluar las pruebas, ejecute `TestSuite("EP04_01.extensión").run()` en una nueva celda, reemplazando la extensión por la del lenguaje utilizado (`.py`, `.java`, `.c`, `.cpp`, `.js` o `.r`). El sistema descarga los casos de prueba de GitHub, ejecuta el programa y calcula la nota automáticamente.

Para probar código Python directamente, sin guardar archivo, use `run_code(codigo)` pasando el código como *string* en una variable `codigo`:

```
codigo = """
from morph import mm
# ... su código aquí ...
"""
TestSuite("EP04_01").run_code(codigo)
```

2.12.1 EP02_01 ⊙ Ajuste de Brillo y Contraste

En esta actividad, el objetivo es implementar un operador de punto para **transformación lineal de intensidad**, aplicando el ajuste dinámico de brillo y contraste en una imagen digital.

2.12.1.1 📋 Directrices de Implementación

El algoritmo debe seguir el flujo de ejecución siguiente:

1. **Dimensiones:** Leer los enteros L (filas) y C (columnas) de la matriz.
2. **Parámetros:** Leer el valor real α (factor de contraste) y el entero β (factor de brillo).

3. **Datos:** Leer los valores enteros de la matriz original.
4. **Mapeo:** Para cada píxel p, calcular el nuevo valor p' mediante la ecuación:

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

5. **Salida:** Mostrar la matriz resultante con las dimensiones $L \times C$.

2.12.1.2 📌 Restricciones Computacionales

- **Redondeo (*Round*):** Se aplica el redondeo matemático al entero más próximo antes de la conversión de tipo.
- **Saturación (*Clipping*):** Los valores deben limitarse al intervalo $[0, 255]$ para preservar el estándar de 8 bits:

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

- **Simulación:** El efecto de los parámetros α y β en la corrección histogramática puede observarse en la Figura 2.12.

2.12.1.3 🧠 Fundamentación Teórica

Las alteraciones modifican el histograma de la imagen para ajustar el perfil de iluminación y la distinción tonal.

Parámetro	Función	Impacto Visual
α (***Alpha***)	Escalar	Modula el **Contraste**. Si $\alpha > 1$, expande el histograma; si $0 \leq \alpha < 1$, lo comprime.
β (***Beta***)	Aditiva	Modula el **Brillo**. Si es positivo, desplaza el histograma hacia la derecha; si es negativo, hacia la izquierda.
clip	Limitador	Restringe el rango dinámico, evitando errores de *underflow* y *overflow*.

2.12.1.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Valores de `alpha` (α) y `beta` (β).
- Líneas siguientes: Elementos numéricos de la matriz original.

Salida:

- Matriz transformada estructurada en L filas y C columnas.

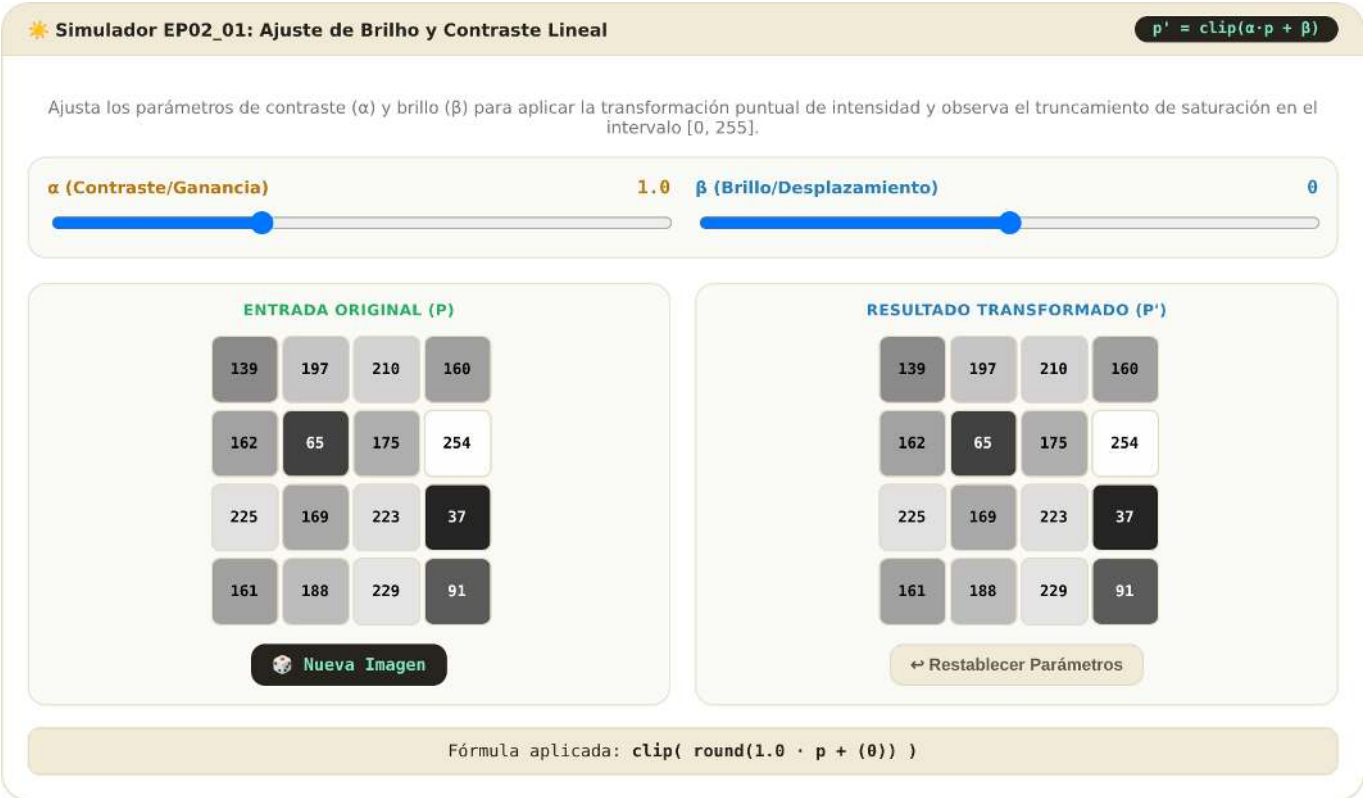
2.12.1.5 📌 Ejemplos

La tabla siguiente presenta un ejemplo práctico del comportamiento esperado del algoritmo, destacando la actuación de los operadores de redondeo y saturación.

Entrada	Salida	Observación
1 4 1.5 -30 0 100 180 255	0 120 240 255	Nótese el efecto de saturación en el último píxel

Ejecutando los Pruebas

Para evaluar las pruebas, ejecute `TestSuite("EP02_01.extensión").run()` en una nueva celda, reemplazando la extensión por la del lenguaje utilizado (`.py`, `.java`, `.c`, `.cpp`, `.js` o `.r`). El sistema descarga los casos de prueba desde GitHub, ejecuta el programa y calcula la nota automáticamente.



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

Figura 2.12: Simulador EP02_01: Ajuste de Brilho y Contraste Lineal (p' = p +)

```
%%writefile EP02_01.py
# tu solución
```

```
Overwriting EP02_01.py
```

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

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

2.12.2 EP02_02 🔬 Submuestreo Espacial

En esta actividad, debes implementar la reducción de la resolución espacial de una imagen mediante el proceso de submuestreo.

- Lee dos enteros **L** y **C**, que representan las dimensiones de la matriz original.
- Lee un valor entero f ($f \geq 1$), que representa el factor de muestreo.

- Lee los valores enteros de la matriz original.
- La nueva imagen debe construirse seleccionando el píxel de la posición $(f \cdot i, f \cdot j)$ de la imagen original.
- Imprime la matriz resultante con las nuevas dimensiones.
- Ver en la Figura 2.13 una simulación de este EP.

📌 **Importante**:

- **Dimensiones Finales:** La imagen muestreada tendrá dimensiones $\lceil L/f \rceil \times \lceil C/f \rceil$. En el contexto de programación, esto equivale al tamaño resultante de un segmentado (slicing) con paso f.
- **Implementación:** No utilices funciones predefinidas de bibliotecas de procesamiento de imágenes (como OpenCV o PIL) para el redimensionamiento. Implementa la lógica de selección de píxeles manualmente o mediante segmentado de matrices.
- **Aliasing:** Ten en cuenta que este proceso puede causar el efecto de *aliasing* (dientes de sierra), donde se pierden detalles finos o aparecen patrones no deseados.

2.12.2.1 🧠 Discretización del Espacio

El submuestreo reduce la resolución espacial de una imagen, seleccionando solo un píxel de cada f píxeles en cada dirección. Es el proceso inverso de la interpolación:

Parámetro	Función	Efecto
Factor f	Salto de muestreo	Define el intervalo de selección. Un factor 2 reduce el ancho y la altura a la mitad.
Resolución	Densidad de píxeles	Disminuye la cantidad total de información espacial de la imagen.
Aliasing	Efecto secundario	Aparición de patrones en escalera o bloques debido a la pérdida de detalles finos.

2.12.2.2 📋 Tarea (especificación para VPL)

Entrada:

La primera línea contiene `L`.

La segunda línea contiene `C`.

La tercera línea contiene el factor `f`.

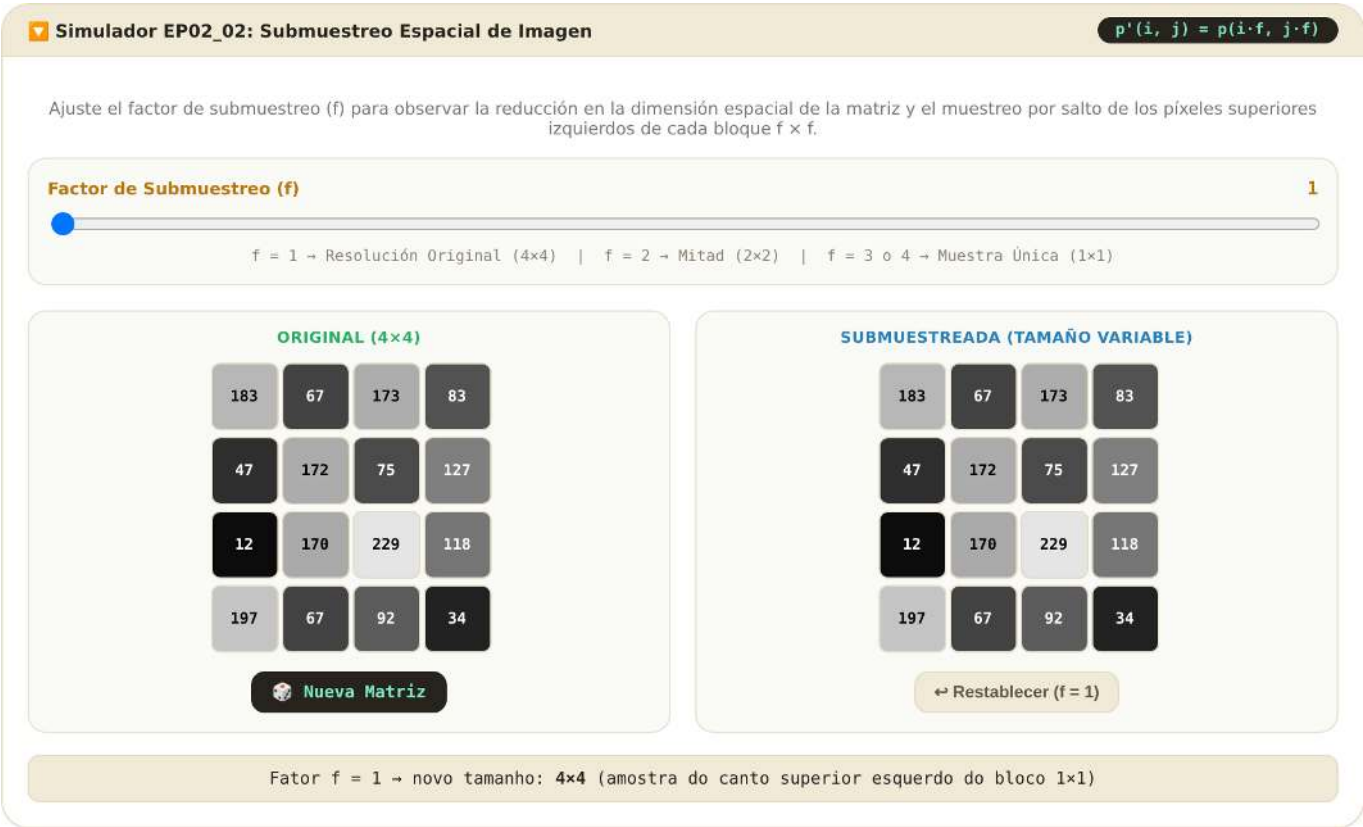
Las líneas siguientes contienen los elementos de la matriz $L \times C$.

Salida:

La matriz reducida con las dimensiones correspondientes al segmentado por `f`.

2.12.2.3 📌 Ejemplos

Entrada	Salida	Observación
2 4 2 10 20 30 40 50 60 70 80	10 30	El factor 2 selecciona los píxeles (0,0) y (0,2) de la primera fila. La segunda fila se ignora.



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

Figura 2.13: Simulador EP02_02: Submuestreo Espacial (Reducción de Resolución por Salto f)

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

```
Overwriting EP02_02.py
```

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

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

2.12.3 EP02_03 🎨 Cuantización de Niveles de Gris

En esta actividad, debes implementar la cuantización uniforme de una imagen, reduciendo la cantidad de niveles de intensidad de gris originales a una nueva escala basada en un número menor de bits.

- Lee dos enteros **L** y **C**, que representan las dimensiones de la matriz.
- Lee un entero k ($1 \leq k \leq 8$), que representa el nuevo número de bits de la imagen.
- Calcula el número de niveles ($N = 2^k$) y el tamaño del intervalo (paso).
- Para cada píxel p, calcula el nuevo valor p' mapeándolo al índice del nivel discretizado correspondiente (que varía de 0 a $2^k - 1$).
- Imprime la matriz resultante con los mismos valores de dimensiones originales.
- Consulta en Figura 2.14 una simulación de este EP.

📌 **Importante**:

- **Posterización:** Al reducir drásticamente los niveles (ej: $k = 2$), notarás que los degradados suaves se transforman en bandas abruptas de color debido a la pérdida de resolución de amplitud.
- **Cálculo del Paso:** El intervalo entre cada nivel se define como $paso = 256/2^k$.

- **Mapeo:** El método de cuantización uniforme por truncamiento que mapea el píxel al índice de su respectivo nivel discretizado está dado por:

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

En términos de implementación (como en Python), esto equivale a la división entera: `p' = p // paso`.

2.12.3.1 🧠 Discretización de la Amplitud

Mientras que el submuestreo trata con la resolución espacial, la cuantización se centra en la precisión del color (amplitud). Reducir los bits significa simplificar la información cromática:

Parámetro	Función	Efecto
Bits (k)	Profundidad de color	Define cuántos tonos diferentes puede tener la imagen (2^k).
Paso	Intervalo de tono	Espaciado entre los niveles de gris permitidos.
Posterización	Fenómeno visual	Transformación de variaciones continuas en bloques de color sólido.

2.12.3.2 📋 Tarea (especificación para VPL)

Entrada:

La primera línea contiene `L`.

La segunda línea contiene `C`.

La tercera línea contiene el número de bits `k`.

Las líneas siguientes contienen los elementos de la matriz $L \times C$.

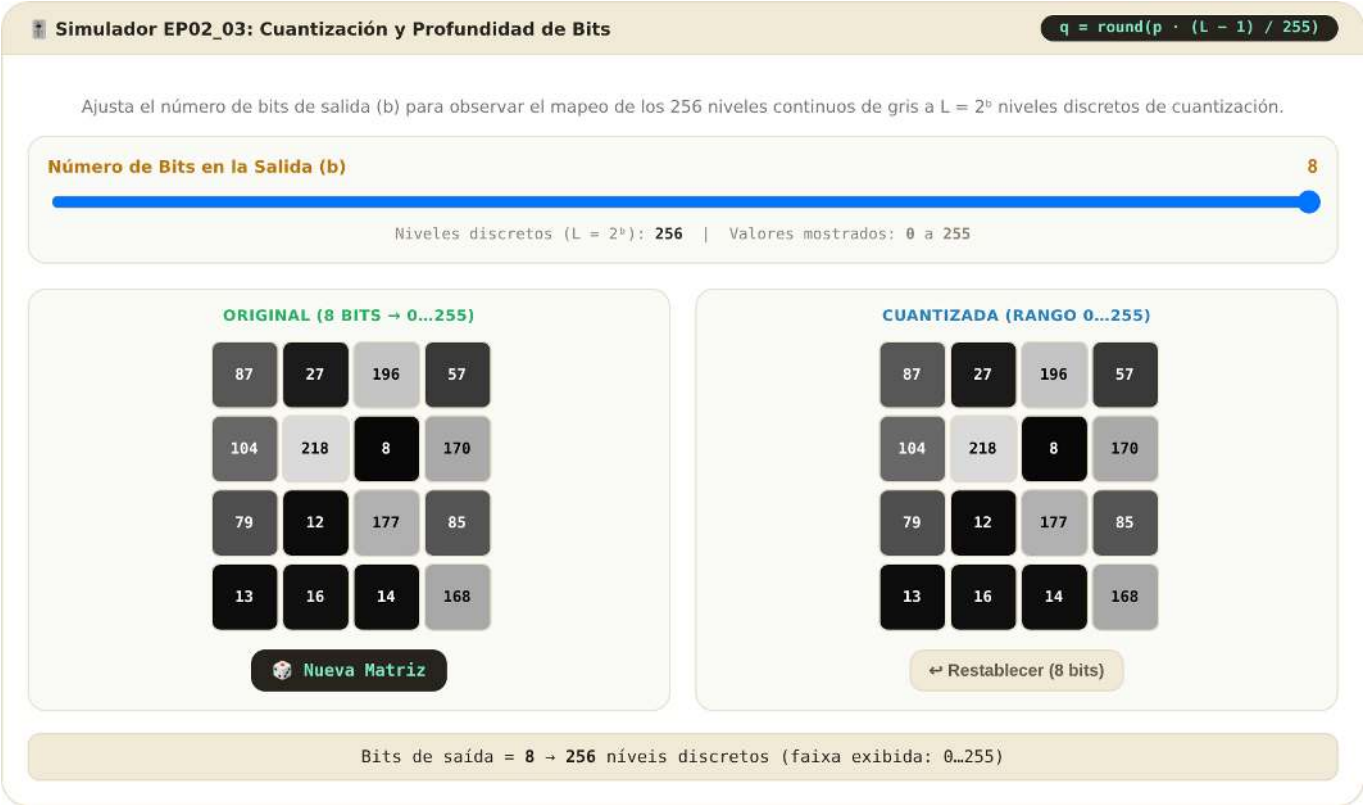
Salida:

La matriz transformada con los índices de los niveles cuantizados, manteniendo el tamaño original $L \times C$.

2.12.3.3 📌 Ejemplos

Entrada	Salida	Observación
1 4 2 0 80 170 255	0 1 2 3	Con $k = 2$, tenemos $2^2 = 4$ niveles discretos disponibles $(0, 1, 2, 3)$. El paso es $256/4 = 64$. Aplicando la división entera por elemento: $0//64 = 0$, $80//64 = 1$, $170//64 = 2$, $255//64 = 3$.
1 5 1 10 50 120 200 250	0 0 0 1 1	Con $k = 1$, tenemos $2^1 = 2$ niveles (0 y 1). Paso $= 256/2 = 128$. Los píxeles menores que 128 resultan en 0, y los píxeles mayores o iguales a 128 resultan en 1.

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



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

Figura 2.14: Simulador EP02_03: Cuantización y Profundidad de Bits (Reducción del Número de Niveles de Gris)

```
Overwriting EP02_03.py
```

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

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

2.12.4 EP02_04 📐 Transformada de Distancia en Imagen Binaria

Dada una imagen binaria donde los píxeles de valor **1** representan el *objeto* y los píxeles **0** representan el *fondo*, la distancia de un píxel de fondo recibe la **menor distancia al píxel de objeto más cercano**. Los píxeles de objeto reciben distancia **0**. Para simplificar, considere que la imagen tiene **solo un único objeto con un píxel de valor 1**.

Problema: Lea una imagen binaria $L \times C$ y una métrica, y calcule esa distancia simplificada aplicando una de las tres fórmulas:

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

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

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

donde Δr es la diferencia de filas y Δc la diferencia de columnas entre dos píxeles.

2.12.4.1 🖼️ ¿Por qué es importante? - Aplicaciones de la DT

La Transformada de Distancia (DT) aparece en decenas de pipelines de visión por computadora:

Métrica	Complejidad	Aplicación típica
Euclidiana	🔴 $O(n^2)$ ingenuo	Esqueletización, emparejamiento de formas
City-block	🟡 $O(n)$ con 2 pasadas	Morfología, dilatación/erosión
Chessboard	🟢 $O(n)$ con 2 pasadas	Morfología, dilatación/erosión

2.12.4.2 📌 Requisitos Técnicos

- **Entrada:** * Primera línea: L y C (enteros).
  - Segunda línea: nombre de la métrica (`euclidean`, `cityblock` o `chessboard`).

  - A continuación, la matriz binaria $L \times C$ (valores 0 o 1).
- **Píxeles de objeto (1):** distancia $= 0$ (o 0.00 para euclidiana).

- **Píxeles de fondo (0):** distancia al único píxel de objeto en la imagen.

- **Redondeo (euclidiana):** imprimir con **2 decimales** (formato `:.2f`). City-block y Chessboard producen enteros — imprimir sin decimales.

- **Salida:** valores separados por espacio, una línea por fila de la matriz.
- Ver en Figura 2.15 una simulación de este EP.

2.12.4.3 📌 Ejemplos

Entrada	Salida	Observación
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	La distancia Chessboard es $\max(\lVert dx \rVert, \lVert dy \rVert)$. El único píxel objeto es $(2, 2) = 0$; los demás almacenan su distancia mínima hasta él.

2.12.4.4 📌 Observaciones finales

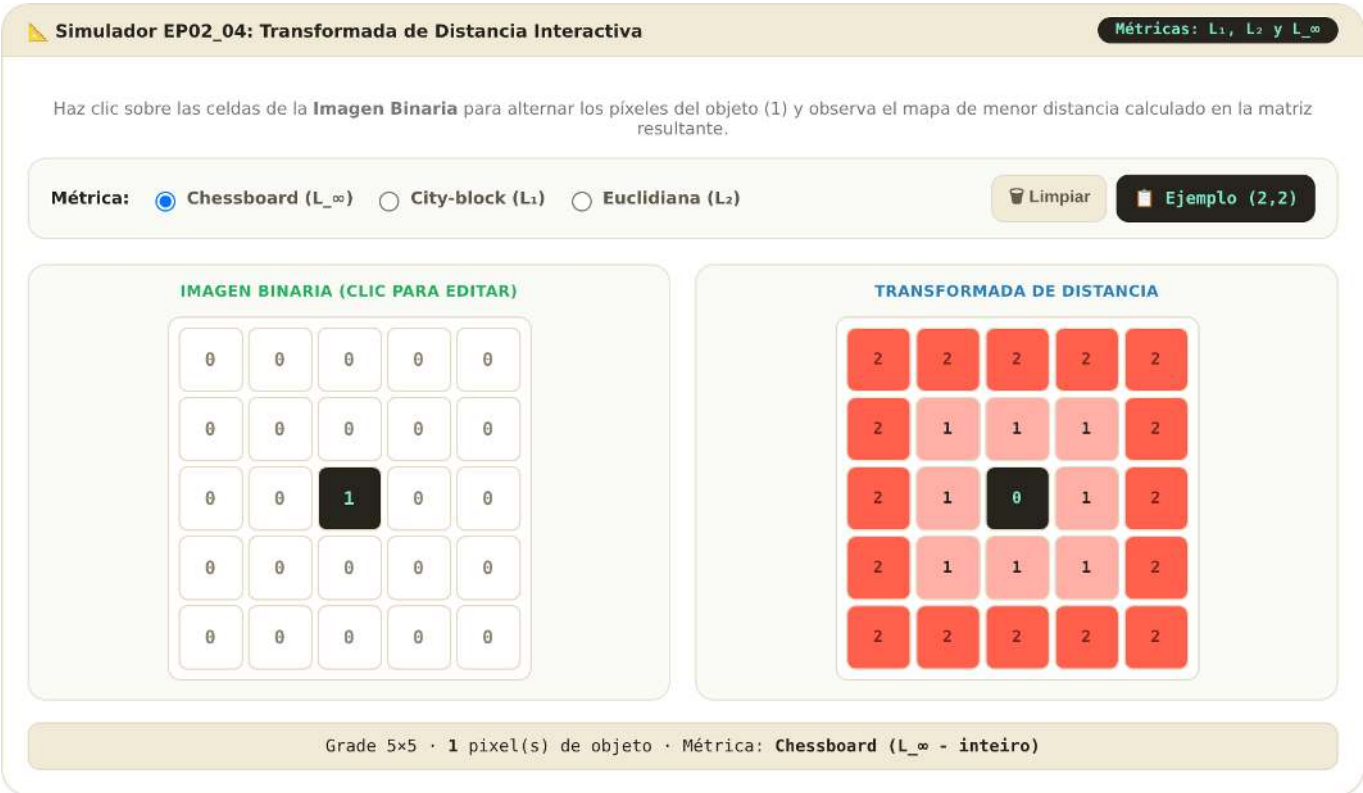
- Como la imagen tiene **solo un objeto de un píxel**, la distancia de cada píxel de fondo es simplemente la distancia de ese píxel al único punto objeto.
- La implementación puede usar fuerza bruta (recorrer todos los píxeles de la imagen y calcular la distancia directamente), ya que L y C son pequeños en los casos de prueba.
- Este problema es un calentamiento para la Transformada de Distancia general, que se trabajará en capítulos posteriores con múltiples objetos y algoritmos optimizados.

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

```
Overwriting EP02_04.py
```

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

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



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

Figura 2.15: Simulador EP02_04: Transformada de Distância em Imagem Binária (Chessboard, City-block y Euclidiana)

2.12.5 EP02_05 Transposición de Imagen

En esta actividad, debes implementar el desplazamiento espacial de una imagen. La traslación mueve cada píxel de la imagen original a una nueva posición basándose en un vector de desplazamiento.

- Lee dos enteros **L** y **C**, que representan las dimensiones de la matriz.
- Lee dos enteros t_x (desplazamiento horizontal) y t_y (desplazamiento vertical).
- Lee los valores enteros de la matriz original.
- Calcula la nueva posición (x', y') para cada píxel (x, y) original.
- Imprime la matriz resultante con las mismas dimensiones que la original.
- Consulta en Figura 2.16 una simulación de este EP.

📌 **Importante**:

- **Relleno:** Los píxeles que "entran" en la imagen debido al desplazamiento y no tienen un correspondiente en la original deben rellenarse con **0** (negro).
- **Descarte:** Los píxeles que, después de la traslación, queden fuera de los límites de la matriz ($0 \dots L-1$ o $0 \dots C-1$) deben ignorarse.
- **Coordenadas:** Considera x como el índice de la fila e y como el índice de la columna.

2.12.5.1 🧠 Desplazamiento Espacial

Trasladar una imagen significa mover todos sus puntos una distancia fija en direcciones especificadas. Matemáticamente, usando coordenadas homogéneas, la operación se describe como:

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

Lo que resulta en las ecuaciones simples:

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

2.12.5.2 📋 Tarea (especificación para VPL)

Entrada:

La primera línea contiene `L`.

La segunda línea contiene `C`.

La tercera línea contiene los enteros `tx` y `ty`.

Las líneas siguientes contienen los elementos de la matriz $L \times C$.

Salida:

La matriz resultante con las mismas dimensiones $L \times C$ después del desplazamiento.

2.12.5.3 📌 Ejemplos

Entrada	Salida	Observación
2 2 1 1 10 20 30 40	0 0 0 10	**Desplazamiento ($t_x = 1, t_y = 1$):** Cada píxel se mueve una posición hacia la derecha (horizontal) y una hacia abajo (vertical). El píxel $(0, 0) = 10$ va al destino $(1, 1)$ (esquina inferior derecha). Las posiciones vacías se rellenan con 0.
3 3 -1 0 1 2 3 4 5 6 7 8 9	2 3 0 5 6 0 8 9 0	**Desplazamiento ($t_x = -1, t_y = 0$):** Cada píxel se mueve una posición hacia la izquierda (horizontal). La primera columna original (`1`, `4`, `7`) se descarta, las demás columnas se mueven hacia la izquierda y la última columna resultante se rellena con ceros (0).

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

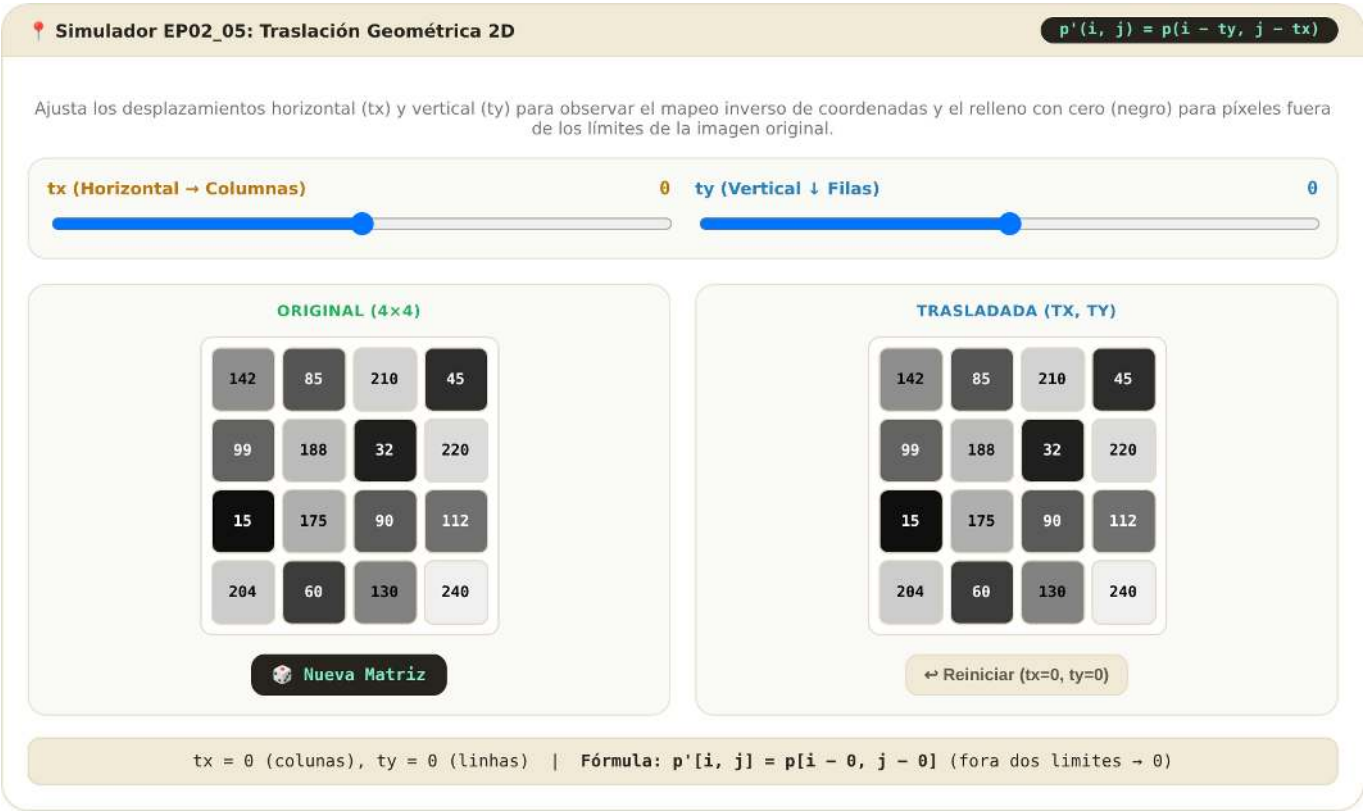
```
Overwriting EP02_05.py
```

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

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

2.12.6 EP02_06 🔄 Rotación de Imagen

En esta actividad, debes implementar la rotación de una imagen alrededor de su centro geométrico. Esta operación requiere el mapeo de coordenadas y el uso de técnicas de interpolación para determinar los nuevos valores de los píxeles.



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

Figura 2.16: Simulador EP02_05: Traslación Geométrica de Imagen (Desplazamiento tx y ty con Relleno de Borde)

- Lea dos enteros **L** y **C**, que representan las dimensiones de la matriz.
- Lea un valor real θ (ángulo en grados) y una *cadena* que representa el **método** de interpolación (`nearest` o `bilinear`).
- Lea los valores enteros de la matriz original.
- Realice la rotación alrededor del centro de la imagen $(L/2, C/2)$.
- Imprima la matriz resultante con las mismas dimensiones que la original.
- Ver en Figura 2.17 una simulación de este EP.

📌 **Importante**:

- **Mapeo Inverso:** Para evitar "huecos" en la imagen final, recorra cada píxel (x', y') de la imagen de destino y calcule su posición correspondiente (x, y) en la imagen original usando la matriz de rotación inversa.

- **Interpolación:**

- `nearest`: Asigna el valor del píxel más cercano a la coordenada calculada.

- `bilinear`: Calcula un promedio ponderado basado en los 4 vecinos más cercanos.

- **Bordes:** Los píxeles cuyo origen (x, y) caiga fuera de los límites de la imagen original deben rellenarse con **0**.

2.12.6.1 🧠 Transformación por Ángulo

La rotación de un punto (x, y) con respecto al origen por un ángulo θ se da mediante la matriz de transformación. Para rotar alrededor de un centro (x_c, y_c), primero trasladamos el centro al origen, rotamos y trasladamos de vuelta:

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

Consejo: Use el mapeo inverso para garantizar que todos los píxeles de la imagen de salida se rellenen correctamente.

2.12.6.2 📋 Tarea (especificación para VPL)

Entrada:

La primera línea contiene `L`.

La segunda línea contiene `C`.

La tercera línea contiene el ángulo `theta` (en grados) y el método `interp` (`nearest` o `bilinear`).

Las líneas siguientes contienen los elementos de la matriz $L \times C$.

Salida:

La matriz rotada con `L` filas y `C` columnas.

2.12.6.3 📌 Ejemplos

Entrada	Salida	Observación
2 2 90 nearest 1 2 3 4	3 1 4 2	Rotación de 90° horaria: la columna 0 se convierte en la fila 0 (de abajo hacia arriba). $(0,0) = 1 \to (1,0)$, $(1,0) = 3 \to (0,0)$, $(0,1) = 2 \to (1,1)$, $(1,1) = 4 \to (0,1)$.
3 3 45 bilinear 0 0 0 0 255 0 0 0 0	0 180 0 180 255 180 0 180 0	Rotación de 45°: el píxel central permanece 255; los vecinos directos reciben un valor interpolado ≈ 180 por bilinear; las esquinas permanecen 0.

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

```
Overwriting EP02_06.py
```

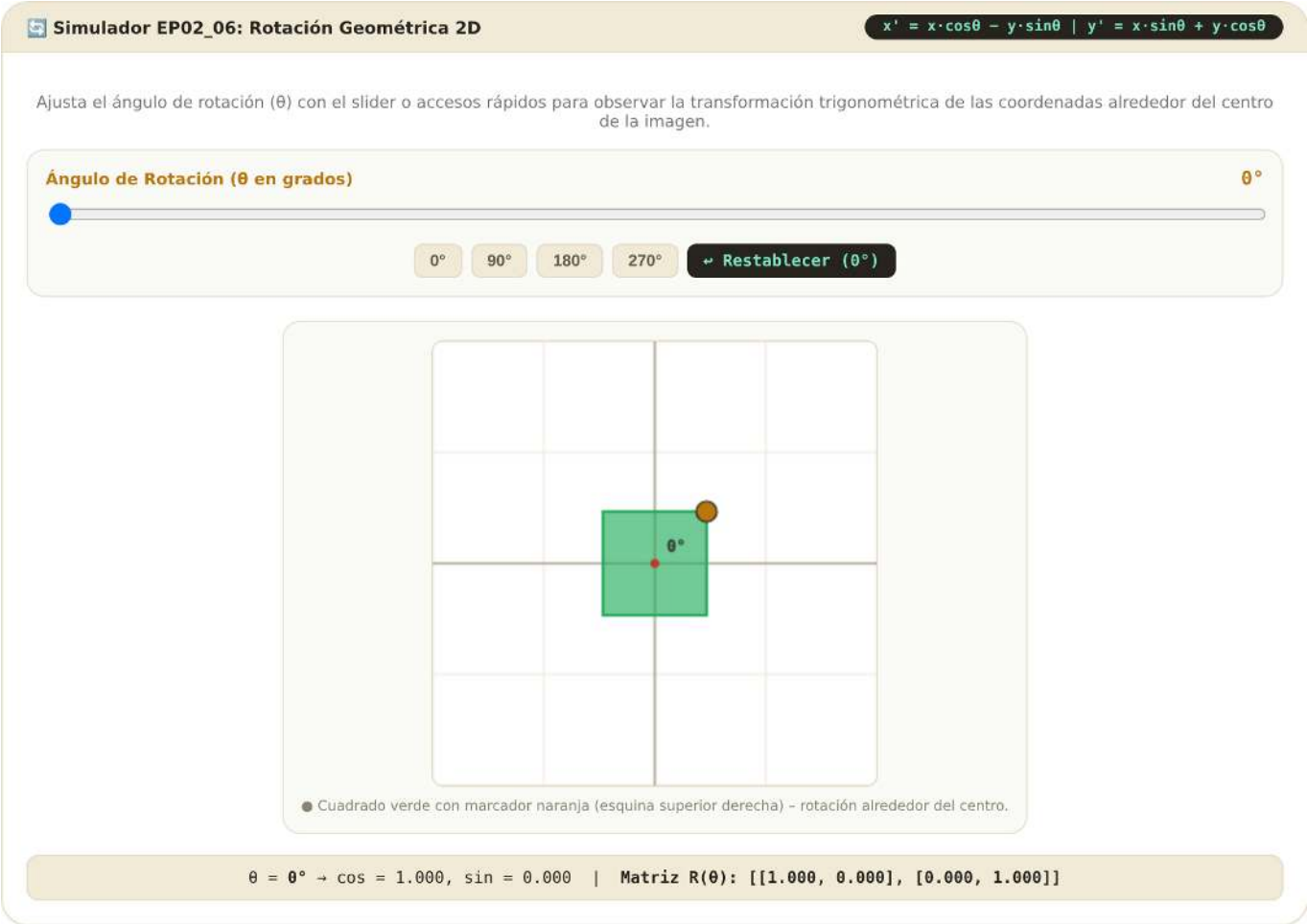
```
TestSuite("EP02_06.py").run()
```

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

2.12.7 EP02_07 🔍 Redimensionamiento (Escala)

En esta actividad, debes implementar el redimensionamiento de una imagen utilizando factores de escala. A diferencia del submuestreo simple, aquí utilizaremos técnicas de interpolación para permitir tanto la ampliación como la reducción de la imagen.

- Lee dos enteros **L** y **C**, que representan las dimensiones de la matriz original.



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

Figura 2.17: Simulador EP02_06: Rotación de Imagen en Torno al Origen por Ángulo

- Lee dos valores reales s_x (escala en las filas) y s_y (escala en las columnas).
- Lee una *cadena* que representa el **método** de interpolación (`nearest` o `bilinear`).
- Lee los valores enteros de la matriz original.
- Calcula las nuevas dimensiones: $L' = \text{round}(L \times s_x)$ y $C' = \text{round}(C \times s_y)$.
- Imprime la matriz resultante con las nuevas dimensiones.
- Ver en Figura 2.18 una simulación de este EP.

📌 **Importante**:

- **Mapeo inverso:** Para cada píxel (x', y') de la imagen de destino, encuentra la posición correspondiente en el origen usando $(x, y) = (x'/s_x, y'/s_y)$.
- **Interpolación:**
- `nearest`: Selecciona el valor del píxel más cercano (redondeo de las coordenadas).
- `bilinear`: Realiza una interpolación lineal doble entre los cuatro píxeles vecinos más cercanos en la imagen original.
- **Bordes:** Asegúrate de que el mapeo no intente acceder a índices fuera del intervalo $[0, L-1]$ y $[0, C-1]$.

2.12.7.1 🧠 Interpolación para Ampliación/Reducción

Redimensionar una imagen por factores (s_x, s_y) exige el relleno de vacíos (en la ampliación) o la fusión de información (en la reducción). El método de interpolación define la calidad visual del resultado:

Método	Funcionamiento	Efecto Visual
Nearest	Toma el valor del vecino más cercano.	Rápido, pero genera efecto "pixelado" o bloques.
Bilineal	Promedio ponderado de los 4 vecinos (2×2).	Suaviza la imagen, reduciendo el aliasing.

2.12.7.2 📋 Tarea (especificación para VPL)

Entrada:

La primera línea contiene `L`.

La segunda línea contiene `C`.

La tercera línea contiene los factores `sx` y `sy`.

La cuarta línea contiene el método `interp` (`nearest` o `bilinear`).

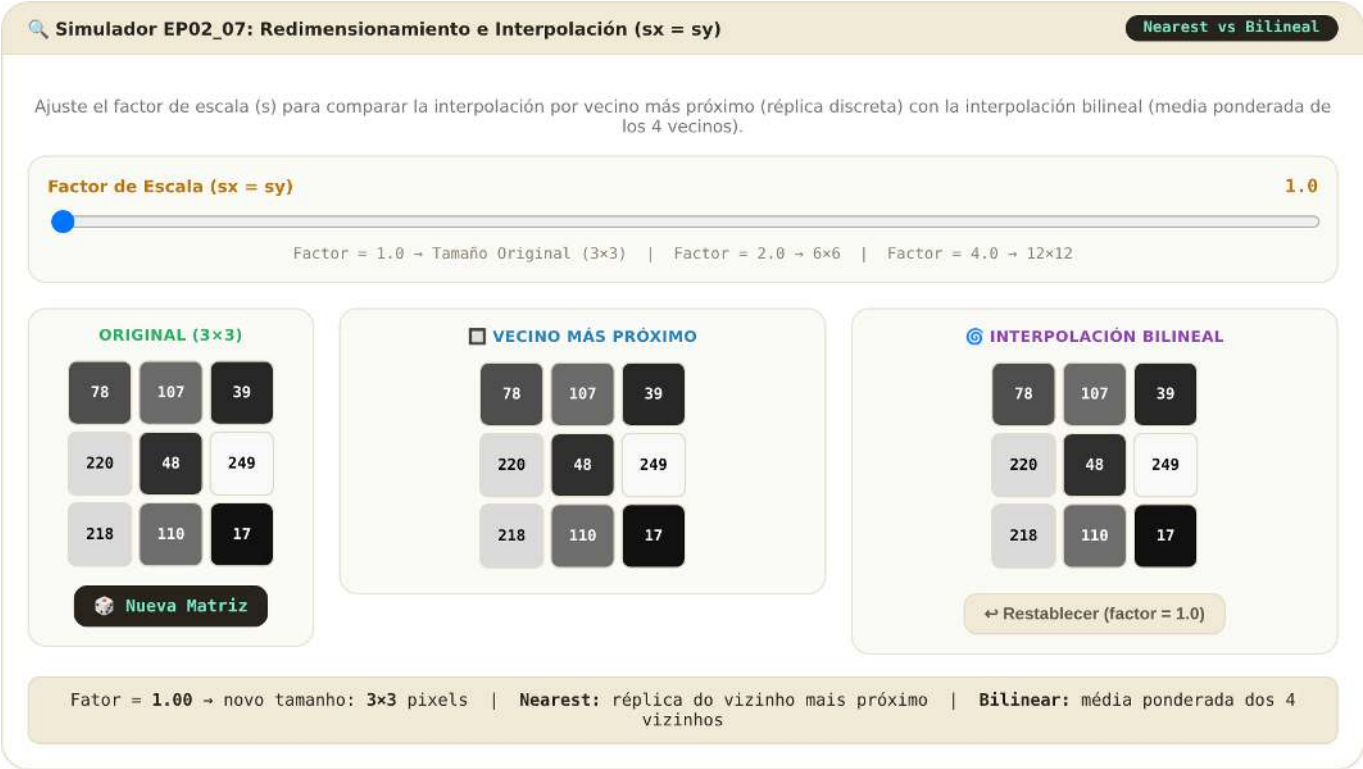
Las líneas siguientes contienen los elementos de la matriz $L \times C$.

Salida:

La matriz redimensionada con dimensiones $L' \times C'$.

2.12.7.3 📌 Ejemplos

Entrada	Salida	Observación
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	Ampliación 2×: cada píxel original se replica en un bloque 2×2. La imagen 2×2 se convierte en 4×4.
2 2 0.5 0.5 nearest 10 20 30 40	10	Reducción 0.5×: la imagen 2×2 se convierte en 1×1. Con nearest, el único píxel de salida muestrea la posición $(0, 0) = 10$.



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

Figura 2.18: Simulador EP02_07: Redimensionamiento Espacial e Interpolación (Vecino más cercano vs Bilineal)

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

# 1. Lectura de las dimensiones, factores y método
l = int(input())
c = int(input())
sx, sy = map(float, input().split())
interp = input().strip()

# 2. Lectura de la imagen original
img = mm.readImg(l, c)
```

```
# 3. Nuevas dimensiones
l_new = round(l * sx)
c_new = round(c * sy)

# 4. Redimensionamiento usando mm.resize
# cv2.resize usa (ancho, alto) = (columnas, filas)
resultado = mm.resize(img, (c_new, l_new), method=interp)

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

```
Overwriting EP02_07.py
```

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

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

2.12.8 EP02_08 🔀 Cortante (Shear)

En esta actividad, debes implementar la transformación de cortante en una imagen. El cortante es una transformación afín que desplaza cada punto en una dirección fija, por un valor proporcional a su distancia de una recta paralela a esa dirección, resultando en un efecto de inclinación.

- Lee dos enteros **L** y **C**, que representan las dimensiones de la matriz.
- Lee dos valores reales sh_x (cortante horizontal) y sh_y (cortante vertical).
- Lee una *cadena* que representa el **método** de interpolación (`nearest` o `bilinear`).
- Lee los valores enteros de la matriz original.
- Aplica la transformación manteniendo el tamaño original de la imagen (recortando lo que exceda los límites).
- Imprime la matriz resultante con las dimensiones $L \times C$.
- Ver en la Figura 2.19 una simulación de este EP.

📌 **Importante**:

- **Mapeo Inverso:** Para cada píxel (x', y') de la imagen de destino, calcula la posición correspondiente en el origen (x, y) utilizando la matriz de cortante inversa.
- **Relleno:** Las coordenadas que resulten en posiciones fuera de la matriz original deben rellenarse con **0**.
- **Coordenadas:** Para fines de esta implementación, considera x como el índice de la fila e y como el índice de la columna.

2.12.8.1 🧠 Distorsión Afín

El cortante altera la geometría de la imagen inclinando sus ejes. La relación entre las coordenadas originales (x, y) y las transformadas (x', y') está dada por:

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

Esto resulta en las ecuaciones:

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

2.12.8.2 📋 Tarea (especificación para VPL)

Entrada:

La primera línea contiene `L`.

La segunda línea contiene `C`.

La tercera línea contiene los factores `shx` y `shy`.

La cuarta línea contiene el método `interp` (`nearest` o `bilinear`).

Las líneas siguientes contienen los elementos de la matriz $L \times C$.

Salida:

La matriz transformada con las mismas dimensiones $L \times C$.

2.12.8.3 📌 Ejemplos

Entrada	Salida	Observación
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	Cortante horizontal: fila i se desplaza $\lfloor i \cdot 0.5 \rfloor$ píxeles. Fila $0 \to 0$px, fila $1 \to 0$px, fila $2 \to 1$px. Los píxeles desplazados hacia fuera se descartan y las posiciones vacías se rellenan con 0.
2 2 0.0 1.0 nearest 10 20 30 40	10 0 30 20	Cortante vertical: columna j se desplaza $\lfloor j \cdot 1.0 \rfloor$ píxeles hacia abajo. Columna $0 \to 0$px (sin cambios), columna $1 \to 1$px: 20 baja a $(1, 1)$ y $(0, 1)$ queda 0.

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

```
Overwriting EP02_08.py
```

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

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

2.12.9 EP02_09 🧩 Transformación Afín Genérica

En esta actividad, debes implementar una transformación afín arbitraria sobre una imagen. Esta operación es la generalización de todas las transformaciones lineales (escala, rotación, cizallamiento) combinadas con la traslación, permitiendo manipulaciones geométricas complejas mediante una única matriz.

- Lee dos enteros **L** y **C**, que representan las dimensiones de la matriz.
- Lee **seis valores reales** (a, b, t_x, c, d, t_y) que componen la matriz de transformación afín 2×3.
- Lee una *cadena* que representa el **método** de interpolación (`nearest` o `bilinear`).
- Lee los valores enteros de la matriz original.
- Aplica la transformación manteniendo el tamaño original $L \times C$.
- Imprime la matriz resultante.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap02/sim-ep0208-cisalhamento.html

Figura 2.19: Simulador EP02_08: Transformación Geométrica de Cortante 2D (Cizallamiento Horizontal y Vertical)

- Ver en Figura 2.20 una simulación de este EP.

📌 **Importante**:

- **Mapeo Inverso:** Para calcular el valor de cada píxel en la imagen de destino, debes utilizar la **inversa** de la matriz de transformación afín proporcionada para encontrar la coordenada correspondiente en la imagen original.
- **Relleno:** Las coordenadas calculadas que caigan fuera de los límites $[0, L-1]$ y $[0, C-1]$ de la imagen original deben resultar en un píxel de valor **0**.
- **Flexibilidad:** Esta implementación debe ser capaz de realizar cualquiera de las tareas anteriores (traslación, rotación, etc.) bastando con alterar los parámetros de la matriz.

Consejo:

```
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 🧠 Combinación de Operaciones

La transformación afín preserva puntos, rectas y planos. En el procesamiento de imágenes, mapea la posición (x, y) a (x', y') siguiendo el sistema:

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

O, de forma compacta en coordenadas homogéneas:

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

2.12.9.2 📋 Tarea (especificación para VPL)

Entrada:

La primera línea contiene `L`.

La segunda línea contiene `C`.

La tercera línea contiene seis flotantes: `a b tx c d ty`.

La cuarta línea contiene el método `interp` (`nearest` o `bilinear`).

Las líneas siguientes contienen los elementos de la matriz $L \times C$.

Salida:

La matriz transformada con las dimensiones originales $L \times C$.

2.12.9.3 📌 Ejemplos

Entrada	Salida	Observación
2 2 1.0 0.0 0.5 0.0 1.0 0.5 bilinear 10 20 30 40	15 20 25 30	Traslación fraccionaria ($t_x = 0.5, t_y = 0.5$): cada píxel de salida (i, j) muestrea la posición $(i + 0.5,\ j + 0.5)$ de la entrada mediante bilinear. Ej: $(0, 0)$ interpola los cuatro vecinos $\rightarrow$ 15.
3 3 2.0 0.0 0.0 0.0 2.0 0.0 nearest 1 2 3 4 5 6 7 8 9	1 1 2 1 1 2 4 4 5	Escala 2× mediante matriz afín ($a = 2, d = 2$): cada píxel de salida (i, j) muestrea la posición $(2i, 2j)$ de la entrada con nearest. Ej: $(0, 2) \rightarrow (0, 4)$ fuera de la imagen $\rightarrow$ nearest recorta a $(0, 2) = 3$… espera confirmación de la lógica de borde.

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

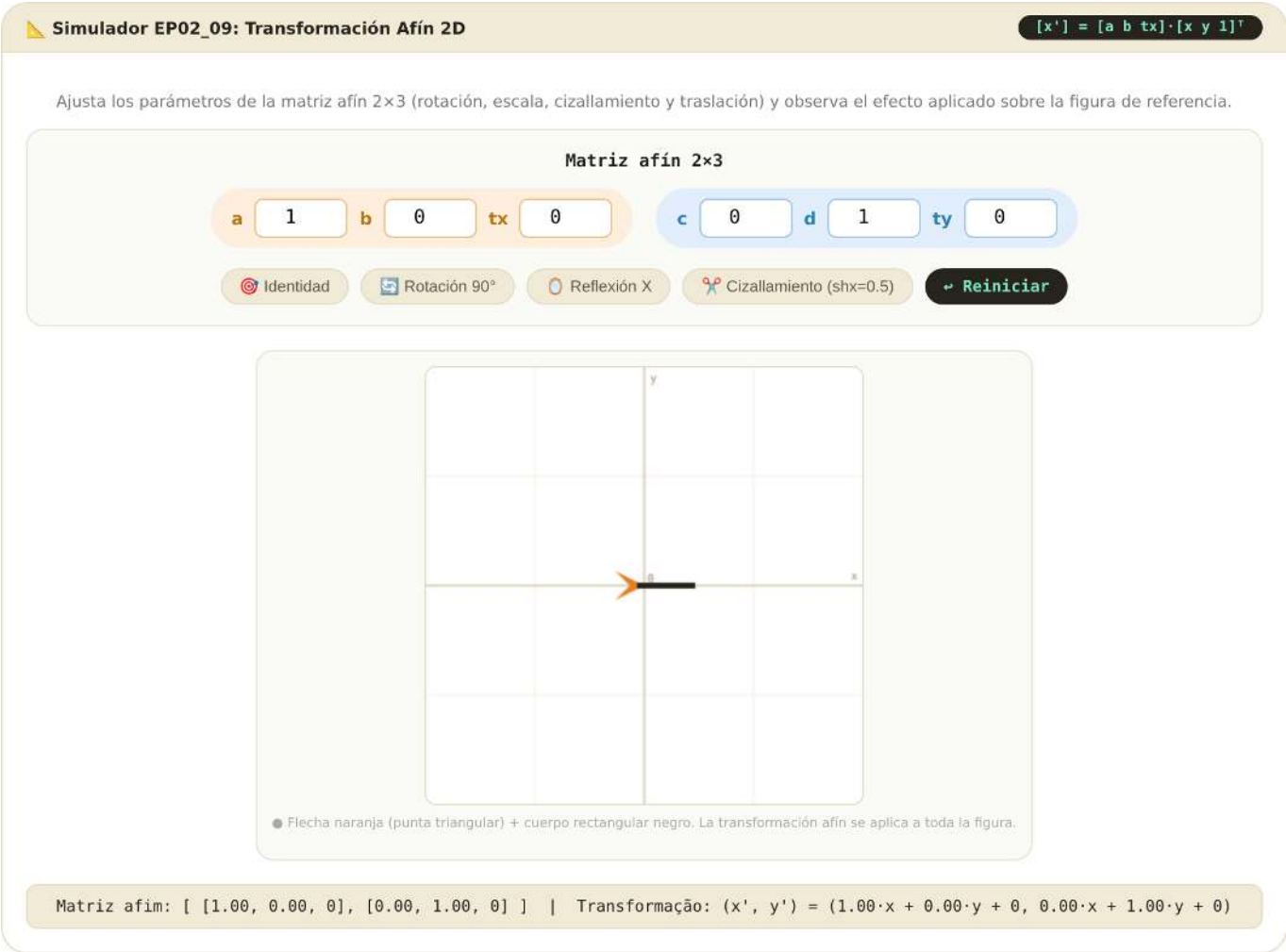
```
Overwriting EP02_09.py
```

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

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

2.12.10 EP02_10 🎯 Corrección de Perspectiva (Homografía)

En esta actividad, debes implementar la transformación de perspectiva, también conocida como homografía. A diferencia de las transformaciones afines, la perspectiva no preserva el paralelismo, permitiendo "rectificar" objetos inclinados, como documentos o placas capturados en ángulos oblicuos.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap02/sim-ep0209-afim.html

Figura 2.20: Simulador EP02_09: Transformación Afín 2D (Matriz 2×3)

- Lee dos enteros **L** y **C**, que representan las dimensiones de la matriz original.
- Lee **cuatro pares de coordenadas** (x, y) que representan las esquinas del cuadrilátero de origen (objeto distorsionado).
- Lee **cuatro pares de coordenadas** (x, y) que representan las esquinas del cuadrilátero de destino (donde se debe mapear el objeto).
- Lee los valores de la matriz original.
- Calcula la matriz de homografía 3×3 y aplica la transformación.
- Imprime la matriz resultante con las dimensiones de salida especificadas.
- Consulta en Figura 2.21 una simulación de este EP.

📌 **Importante**:

- **Grados de libertad:** La homografía posee 8 grados de libertad (el noveno elemento de la matriz 3×3 es una constante de normalización, generalmente 1), lo que requiere al menos 4 puntos correspondientes para calcularse.
- **Proyección:** Después de multiplicar las coordenadas por la matriz, es necesario dividir los resultados x' e y' por la componente homogénea w para regresar al plano 2D.
- **Uso de bibliotecas:** Para esta tarea, puedes utilizar las funciones `cv2.getPerspectiveTransform` para obtener la matriz y `cv2.warpPerspective` para aplicar la transformación, o implementar el sistema lineal y el mapeo inverso manualmente para un desafío adicional.

```
# Dimensiones de salida: bounding box de los puntos destino + 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))
# o
dst = mm.perspective_transform(img, pts1, pts2, size=(w, h))
```

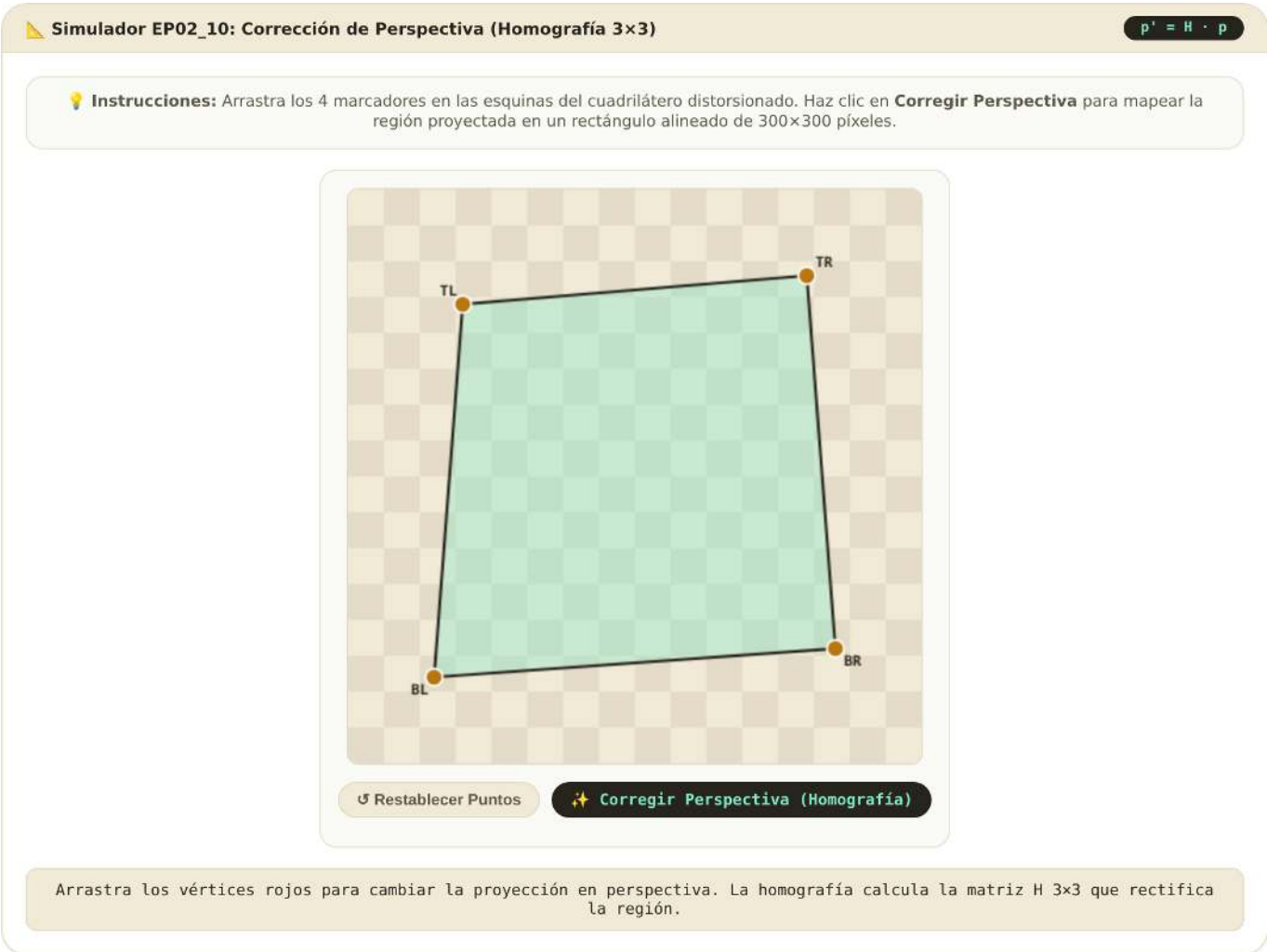
2.12.10.1 🧠 Deformación no afín

Mientras que las transformaciones afines mapean paralelogramos en paralelogramos, la homografía mapea cualquier cuadrilátero en otro cuadrilátero. Esto es esencial para la visión por computadora:

Operación	Característica	Aplicación típica
Homografía	Proyección en plano	Corrección de documentos, escaneo de placas.
Punto de fuga	Convergencia de líneas	Reconstrucción 3D a partir de imágenes 2D.
Warping	Deformación de malla	Estabilización de video y panoramas (stitching).

2.12.10.2 📌 Ejemplos

Entrada	Salida	Observación
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	Las 4 primeras líneas después de las dimensiones son los puntos de origen; las 4 siguientes son los destinos. Con puntos idénticos, la transformación de perspectiva es la identidad y la imagen se preserva.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap02/sim-ep0210-perspectiva.html

Figura 2.21: Simulador EP02_10: Corrección de Perspectiva (Transformación de Homografía 3×3)

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

```
Overwriting EP02_10.py
```

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

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

2.12.11 EP02_11 🏆 Corrección de Perspectiva (Homografía) en Imagen Real

En esta actividad, el objetivo es aplicar la transformación de perspectiva (homografía) para "rectificar" un objeto inclinado en una fotografía real. Trabajarás con la imagen de un periódico, donde la cuadrícula de un juego de Sudoku está distorsionada debido al ángulo en que se tomó la foto.

Tu programa debe leer los parámetros de entrada desde el terminal, cargar la imagen, calcular la matriz de homografía 3×3, aplicar la transformación geométrica y mostrar un indicador global de validación.

- Lee dos enteros **L** y **C**, que representan las dimensiones de filas y columnas (alto y ancho) que debe tener la imagen rectificada de salida.
- Lee **cuatro pares de coordenadas** (x, y) desde el terminal, que representan las cuatro esquinas del cuadrilátero de origen (el Sudoku distorsionado en la imagen original).
- Calcula automáticamente los **cuatro pares de coordenadas de destino** utilizando las dimensiones L y C proporcionadas, mapeando las esquinas a los extremos de la nueva imagen: $(0, 0)$, $(C - 1, 0)$, $(0, L - 1)$ y $(C - 1, L - 1)$.
- Carga la imagen local `sudoku.png` y conviértela a escala de grises.
- Calcula la matriz de homografía y aplica la transformación espacial en la imagen.
- **Salida:** Calcula e imprime la **suma de todos los píxeles** de la imagen resultante.

📌 **Importante**:

- **Archivo de entrada:** La imagen `sudoku.png` debe estar en la misma carpeta que el script. El programa debe leerla directamente del disco (por ejemplo, usando `mm.read("sudoku.png")` o `cv2.imread`).
- **Orden de los puntos:** Asegúrate de que la lectura de los 4 puntos de origen y la generación de los 4 puntos de destino sigan rigurosamente el mismo orden de las esquinas: Superior-Izquierda (TL), Superior-Derecha (TR), Inferior-Izquierda (BL) e Inferior-Derecha (BR).
- **Dimensiones en OpenCV:** Recuerda que funciones como `cv2.warpPerspective` esperan el tamaño de la imagen de salida en el formato `(ancho, alto)`, lo que equivale a `(C, L)`.
- **Interpolación:** Para garantizar la consistencia matemática de la suma de píxeles con el corrector automático, utiliza la interpolación bilineal estándar (`flags=cv2.INTER_LINEAR`).
- **Créditos:** La imagen utilizada es **"Sudoku en periódico"** de Héctor Rodríguez, bajo licencia **CC BY 2.0**.

2.12.11.1 🧠 Contexto del Problema

La homografía tiene 8 grados de libertad, lo que requiere al menos 4 correspondencias de puntos para calcularse. A diferencia de las transformaciones afines, mapea cualquier cuadrilátero en otro cuadrilátero, permitiendo que las líneas que convergen en puntos de fuga vuelvan a ser paralelas:

Operación	Característica	Aplicación Típica
Homografía	Proyección entre planos	Rectificación de documentos, escaneo de placas y códigos QR.
Mapeo Inverso	Recorrido del destino al origen	Evita "agujeros" o píxeles vacíos en la imagen final rectificada.
Warping	Remuestreo espacial	Corrección de distorsión de lentes y montaje de panoramas (*stitching*).

2.12.11.2 📌 Ejemplos

Entrada	Salida	Observación
500 500 100 120 420 95 80 440 450 460	32982820	Las dos primeras entradas son las dimensiones de salida (L y C). Las 4 líneas siguientes son las coordenadas (x, y) de las esquinas del Sudoku en la imagen original + PAD. La salida es la suma total de los píxeles de la imagen rectificada.
200 200 100 120 420 95 80 440 450 460	5277150	Mismos puntos de origen que el ejemplo anterior, pero generando una imagen de salida más pequeña (200×200). La suma de píxeles se reduce proporcionalmente debido a la escala.

2.12.11.3 Adquisición de la imagen del sudoku y conversión a niveles de gris

La Figura 2.22 muestra la lectura de la imagen original seguida de la conversión a tonos de gris y el redimensionamiento a una matriz de 500×500 píxeles, preparando los datos para la etapa siguiente.

La corrección de perspectiva, aplicada en la Figura 2.23 mediante la matriz de homografía, elimina las deformaciones causadas por el ángulo de la cámara y produce una vista frontal y regular de la cuadrícula del Sudoku.

Figura 2.22: Adquisición de la imagen de un Sudoku a la izquierda. A la derecha, conversión a tonos de gris y redimensionamiento. Crédito: Héctor Rodríguez de Guardamar, España (CC BY 2.0).

```
import cv2
import numpy as np

# --- 1. Carga la imagen guardada (sudoku.png) ---
img = mm.read("sudoku.png")                # BGR, 500×500
```

```
# --- 2. Padding para no recortar vértices ---
PAD = 60
img_pad = cv2.copyMakeBorder(
    img, PAD, PAD, PAD, PAD,
    cv2.BORDER_CONSTANT, value=[255, 255, 255]
)

# --- 3. Puntos de origen (esquinas de la cuadrícula en la imagen expandida) ---
pts1 = np.float32([
    [100, 160],    # TL
    [390, 45],    # TR
    [200, 580],    # BL
    [570, 420],    # BR
])
#     W    H

# --- 4. Puntos de destino (vista frontal 500×500) ---
SIZE = 500
pts2 = np.float32([
    [0,    0],
    [SIZE, 0],
    [0,    SIZE],
    [SIZE, SIZE],
])

# --- 5. Homografía y rectificación ---
img_rect = mm.perspective_transform(img_pad, pts1, pts2, size=(SIZE, SIZE))

# --- 6. Visualización ---
mm.show(
    [img_pad, img_rect],
    titles=["Original (con padding)", "Vista frontal rectificada"],
    cols=2, figsize=(10, 6), axis=True
)
```

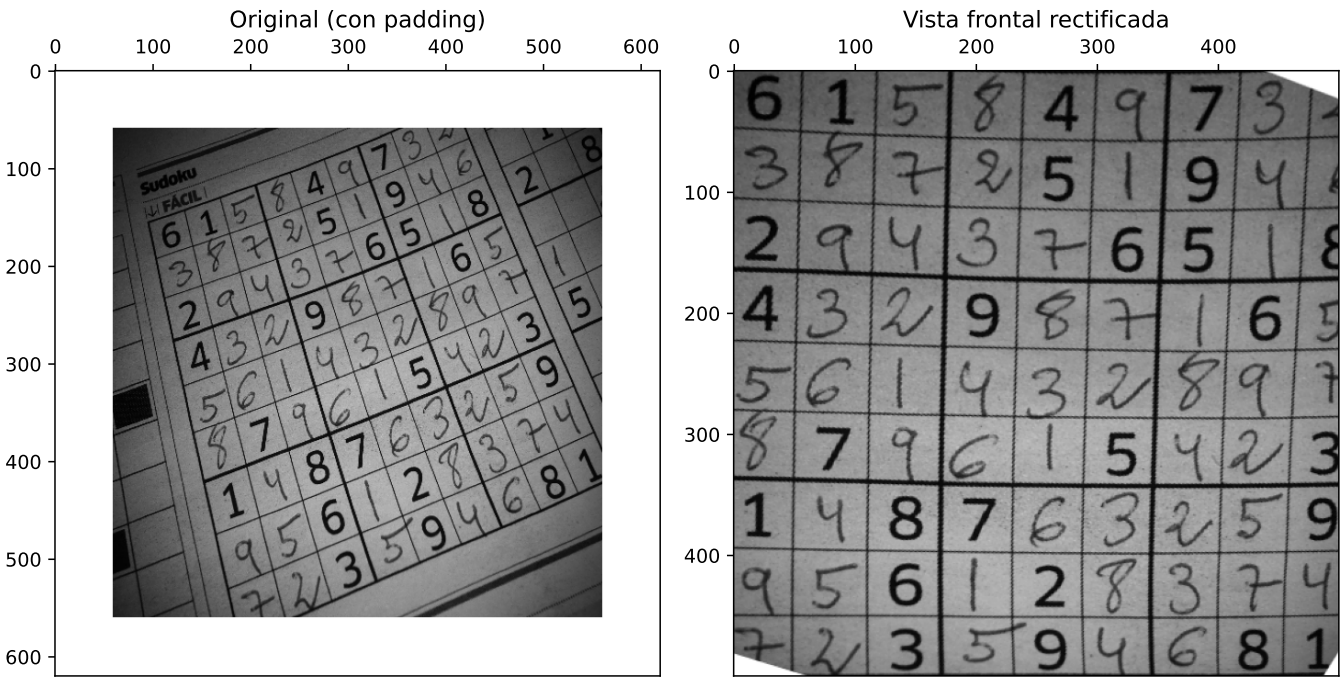
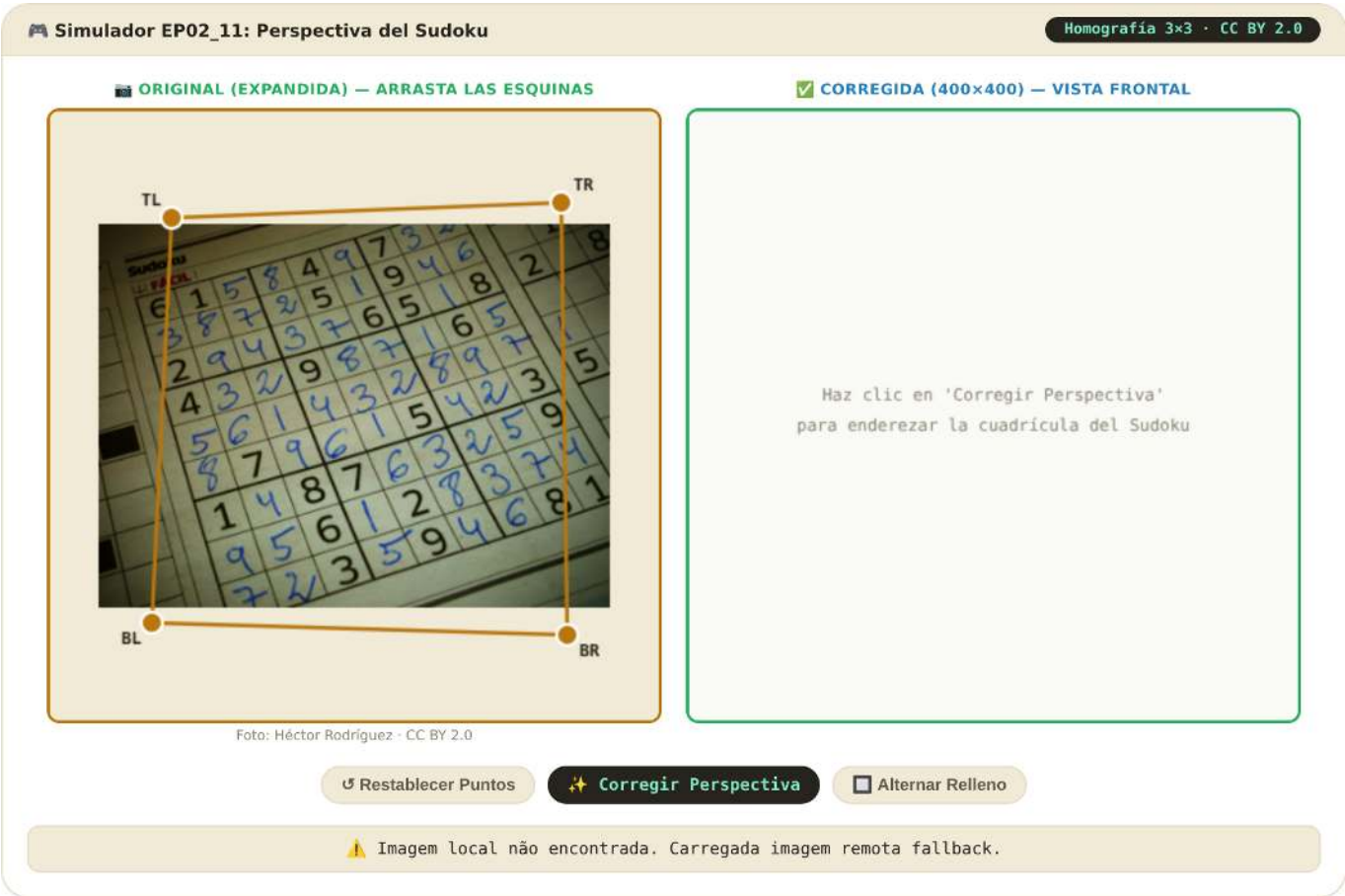


Figura 2.23: Corrección de perspectiva: original y vista frontal rectificada.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap02/sim-ep0211-sudoku.html

Figura 2.24: Simulador EP02_11: Corrección de Perspectiva del Sudoku (Homografía 3×3 con Remuestreo Bilineal)

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

```
Overwriting EP02_11.py
```

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

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

Capítulo 3

Operaciones Espaciales: Intensidad, Histograma y Filtrado

Executar Colab Abrir GitHub

Este capítulo profundiza en el procesamiento de imágenes en el dominio espacial, partiendo de la manipulación directa de píxeles e histogramas para el realce de contraste, hasta la aplicación de filtros locales por convolución para suavizado, reducción de ruido y detección de bordes. El objetivo es desarrollar la intuición matemática y computacional que sustenta gran parte de los algoritmos modernos de Visión Computacional.

3.1 Objetivos

Al final de este capítulo, usted será capaz de:

- **Manipular intensidad y píxeles:** Ejecutar operaciones aritméticas saturadas (`mm.addm`, `mm.subm`) y lógicas bit a bit (`mm.band`, `mm.bor`, `mm.bnot`) para combinación y selección de regiones de interés (ROI), y aplicar *alpha blending* (`mm.blend`) para fusión ponderada de imágenes;
- **Procesar histogramas:** Interpretar el histograma como diagnóstico tonal y aplicar ecualización global mediante CDF (`mm.equalize`); en la ruta Python, también la ecualización adaptativa (CLAHE) y la especificación de histograma para transferencia de perfil tonal entre imágenes;
- **Comprender fundamentos espaciales:** Entender vecindad, *padding* de borde (`mm.pad`) y la diferencia entre correlación cruzada (`mm.conv`) y convolución — incluyendo por qué *kernels* asimétricos como el de Sobel producen resultados distintos en las dos operaciones;
- **Aplicar filtrado de suavizado:** Usar el filtro de media (`mm.blur`, o `mm.conv` con *kernel* uniforme) y el filtro Gaussiano (`mm.gaussian`) para reducción de ruido, comprendiendo la ventaja de la ponderación radial y de la separabilidad Gaussiana;
- **Aplicar filtrado de realce:** Usar el Laplaciano w_4 y w_8 (`mm.laplacian`) para realce isotrópico de bordes, el operador de Sobel (`mm.sobel`) para la magnitud del gradiente — y, en la ruta Python, la descomposición direccional G_x, G_y y ángulo —, y el *Unsharp Masking* (`mm.usm`) para amplificación de alta frecuencia controlada por el parámetro k;
- **Utilizar filtros de orden:** Aplicar el filtro de la mediana (`mm.median`) para eliminación de ruido sal y pimienta, comprendiendo por qué su naturaleza no lineal y la robustez a *outliers* lo hacen superior a los filtros lineales en ese escenario;
- **Resolver problemas prácticos:** Encadenar técnicas en *pipelines* de preprocesamiento (ecualización → Gaussiano → Canny; con CLAHE en lugar de la ecualización en la ruta Python) y usar las funciones de `morph` (`mm.conv`, `mm.histImg`, `mm.equalize`, `mm.drawImgKernel`) para análisis y visualización didáctica de cada etapa.

3.2 Operaciones a Nivel de Intensidad

El nivel más elemental de procesamiento de imágenes actúa directamente sobre los valores de los píxeles, sin considerar la vecindad. Estas operaciones —llamadas **transformaciones de punto** (*point operations*)— son las más rápidas computacionalmente y forman la base para técnicas más complejas.

Formalmente, una transformación de punto puede describirse como:

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

donde $f(x, y)$ es la imagen de entrada, $g(x, y)$ es la salida y T es una función aplicada a cada píxel individualmente.

3.2.1 Preparando el Entorno Práctico

El siguiente bloque carga la biblioteca `morph` del repositorio (el módulo `morph.py` y, en la ruta C++, también el `morph.hpp` utilizado en el `#include` de las celdas compiladas).

```
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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0
```

Como objeto de estudio a lo largo de este capítulo, utilizaremos las imágenes de vida silvestre presentadas en las Figura 3.1 y Figura 3.3.. A partir de ellas, exploraremos operaciones espaciales sobre intensidad, histogramas y filtrado, analizando sus efectos en el realce, la suavización, la reducción de ruido y la detección de bordes, con el fin de comprender los fundamentos matemáticos y computacionales del PDI.

```
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)
```

Figura 3.1: *Mandrill* (*Mandrillus sphinx*) fotografiado en ambiente natural en Sudáfrica. Crédito: Carlos Guilherme Rodrigues (CC BY-SA 3.0).

3.2.2 Operaciones Aritméticas

Las operaciones aritméticas entre imágenes se utilizan ampliamente en PDI para combinar, comparar o realzar información. La **resta de imágenes** es especialmente poderosa para detectar diferencias entre dos cuadros — por ejemplo, en la eliminación del fondo estático en cámaras de vigilancia:

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

La **suma saturada** limita el resultado al intervalo $[0, 255]$: los valores superiores a 255 se fijan en 255, evitando el *overflow* silencioso del tipo `uint8` (p. ej., $200 + 100 = 44$ en lugar de 300). La **resta saturada** aplica el mismo principio por el lado inferior: los valores negativos se fijan en 0.

> ⚠ Saturación y *overflow*
>
> Las operaciones aritméticas en `uint8` sufren *overflow* silencioso: $200 + 100 = 44$ (no 300). `mm.addm` y `mm.subm` realizan la **saturación automática**, fijando el resultado en $[0, 255]$. El *blending* utiliza pesos fraccionarios: `mm.blend` opera internamente en punto flotante y solo entonces redondea y satura a `uint8`.

La Figura 3.2 demuestra la suma de una constante (aclarado) y la resta de una constante (oscurecimiento con saturación en 0).

```
fundo = 60

img_add = mm.addm(img_gray, fundo)
img_sub = mm.subm(img_gray, fundo)

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

Figura 3.2: Operaciones aritméticas saturadas: suma de constante (aclarado) y resta de constante (oscurecido con saturación en 0).

3.2.3 Mezcla Ponderada (*Alpha Blending*)

La **mezcla ponderada** (*alpha blending*) combina dos imágenes utilizando pesos complementarios α y $(1-\alpha)$:

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

Cuando $\alpha = 1$, se obtiene únicamente la imagen f_1; cuando $\alpha = 0$, solo f_2. Los valores intermedios producen una transición suave entre ambas, siendo ampliamente utilizados en composición de imágenes, superposición de capas, marcas de agua y efectos de fusión visual.

Para que la combinación produzca un resultado coherente, es necesario alinear previamente las regiones de interés. En la Figura 3.4, se recorta el rostro del leopardo con `mm.crop(img_leop_gray, 250, H-300, 100, W-200)` y la región facial del mandril con `mm.crop(img_gray, 100, 400, 380, 530)`, de modo que los ojos y la estructura facial queden aproximadamente alineados. El recorte del leopardo se redimensiona entonces (`mm.resize`) a las dimensiones del mandril antes de la mezcla.

`mm.blend` realiza la operación en punto flotante — evitando *overflow* en los cálculos con pesos fraccionarios — y solo entonces redondea y satura el resultado a `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)
```

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

```
# recortes alineados: rostro del leopardo y región facial del mandril
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
)
```

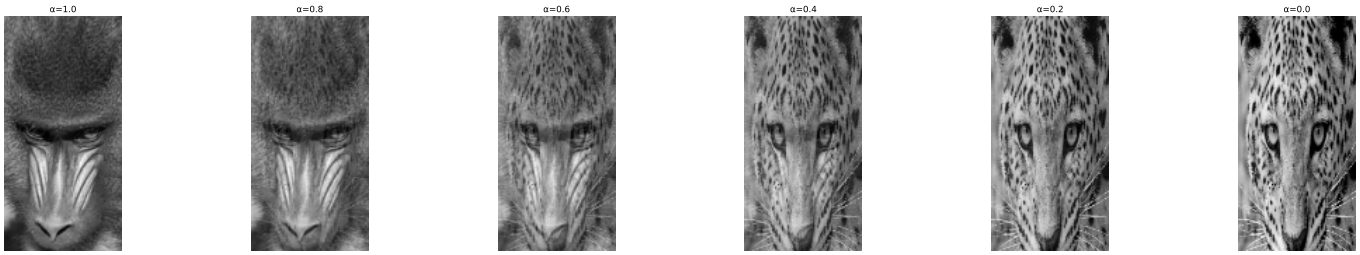


Figura 3.4: *Mezcla alfa* entre recortes alineados de mandril y del leopardo (Figura 3.3) para diferentes valores de . En =1 se ve solo el mandril; en =0, solo el leopardo; valores intermedios fusionan las miradas de las dos imágenes proporcionalmente.

3.2.4 Operaciones Lógicas y Máscaras Bit a Bit

Las operaciones lógicas bit a bit (AND, OR y NOT) actúan directamente sobre los bits de cada píxel y son la base para la creación y aplicación de **máscaras** (*masks*) —imágenes binarias con solo 0 (negro) y 255 (blanco) utilizadas para aislar **Regiones de Interés** (**ROI**).

El comportamiento de cada operación se deriva de la representación binaria del 255 (`11111111`) y del 0 (`00000000`):

- **AND** con la máscara: donde $m = 255$, los bits originales se preservan; donde $m = 0$, el píxel se pone a cero. Resultado: recorte de la ROI.

$$g(x, y) = f(x, y) \text{ AND } m(x, y) \tag{3.4}$$

- **OR** con la máscara: donde $m = 255$, el píxel se fuerza a blanco; donde $m = 0$, se mantiene el valor original. Resultado: iluminación de la ROI.
- **NOT** (sin máscara): invierte todos los bits ($g = 255 - f$), produciendo el negativo fotográfico de la imagen.

La Figura 3.5 ilustra las tres operaciones aplicadas a la imagen del mandril con una máscara circular.

```
h, w = img_gray.shape

# Máscara circular rellena, centrada en la imagen (mm.circle dibuja el
# disco; la versión didáctica, prueba de radio píxel a píxel, es mm.circle0).
mask_circ = mm.circle(np.zeros((h, w), dtype=np.uint8),
                      (w // 2, h // 2), min(h, w) // 3 - 10, 255, -1)

# Operaciones vía morph
img_not = mm.bnot(img_gray)            # NOT: negativo fotográfico
img_and = mm.band(img_gray, mask_circ) # preserva solo la ROI circular
img_or  = mm.bor(img_gray, mask_circ)  # ilumina la región de la máscara

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

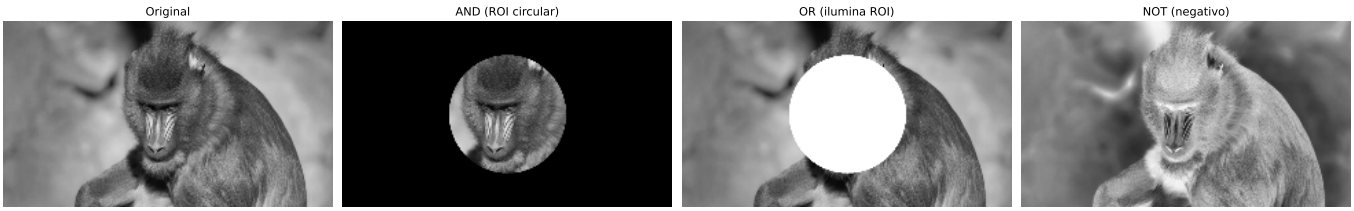


Figura 3.5: Operaciones lógicas bit a bit con máscara circular: AND (aislamiento de la ROI), OR (iluminación de la ROI) y NOT (negativo).

3.3 Histograma de Imágenes

El **histograma** de una imagen en tonos de gris es una función discreta que describe la distribución de frecuencias de las intensidades:

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

donde r_k es el k-ésimo nivel de intensidad, n_k es el número de píxeles con esa intensidad y L es el total de niveles (típicamente 256 para 8 bits). El histograma normalizado estima la probabilidad de cada nivel:

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

donde MN es el total de píxeles. Por ser una **estadística global**, el histograma no contiene información posicional, pero revela características esenciales como brillo medio, contraste y distribución tonal. En la

práctica, `mm.hist(img)` devuelve el vector de conteos $h(r_k)$, que sirve tanto para visualización (mediante `mm.histImg`) como para cálculos como **función de distribución acumulada (CDF)** y ecualización.

> **Interpretación del Histograma**
>
> - **Estrecho a la izquierda:** imagen subexpuesta (oscura).
> - **Estrecho a la derecha:** imagen sobreexpuesta (clara).
> - **Concentrado en el centro:** bajo contraste.
> - **Distribuido por todo el rango:** alto contraste, buena utilización de los tonos disponibles.

La Figura 3.6 presenta el histograma de la imagen del mandril, así como versiones oscurecida (`mm.subm`) y aclarada (`mm.addm`). Se observa el desplazamiento de la distribución de intensidades hacia la izquierda y hacia la derecha, respectivamente. Nótese que el intervalo representado en el eje x no corresponde necesariamente a todo el rango de 0 a 255.

```
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", "Oscurecida (-80)", "Aclarada (+80)",
            "Histograma - original", "Histograma - oscurecida", "Histograma - aclarada"],
    cols=3
)
```

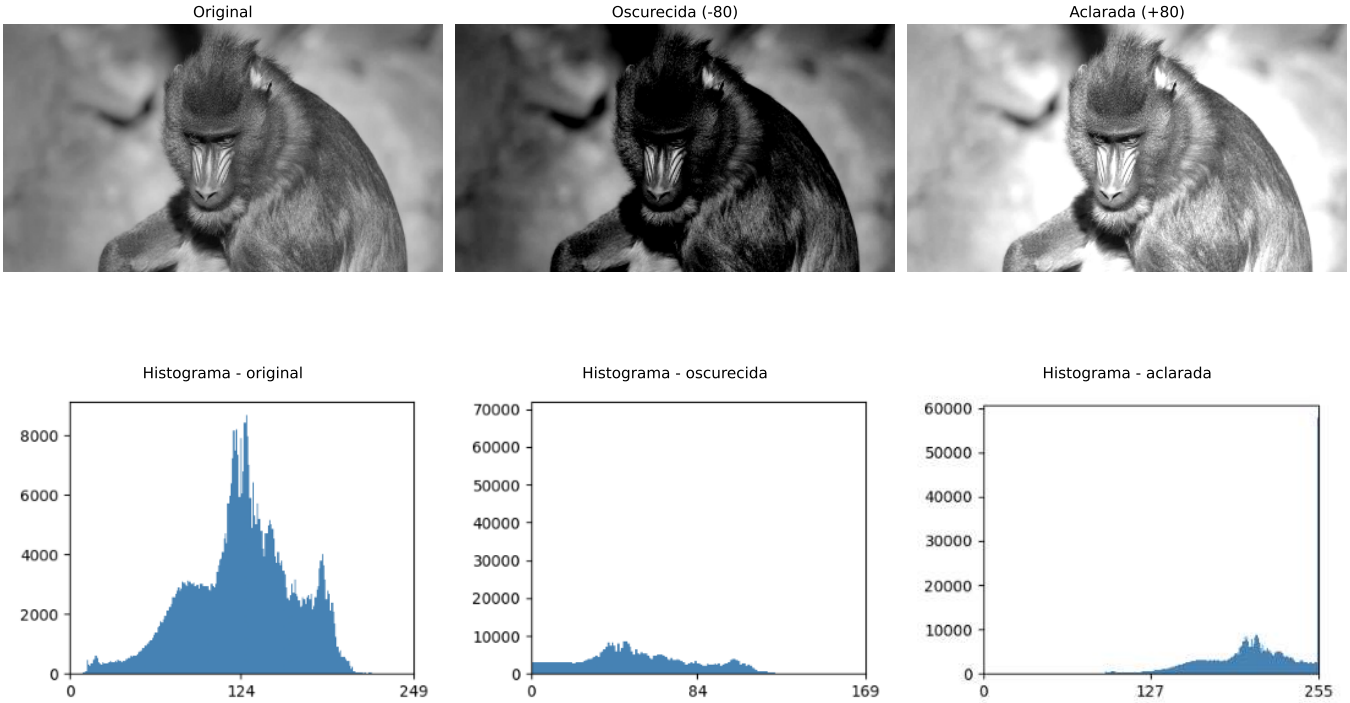


Figura 3.6: Histogramas de la imagen original, de una versión oscurecida (-80) y de una aclarada ($+80$). La resta/suma satura en 0 y 255.

3.3.1 Ecualización de Histograma

La **ecualización de histograma** redistribuye las intensidades para que el histograma resultante sea lo más uniforme posible. El mapeo está dado por la **función de distribución acumulada (CDF)**:

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p(r_j) = \frac{L-1}{MN}\sum_{j=0}^{k} n_j \tag{3.7}$$

La transformación es monótona: los niveles frecuentes reciben intervalos mayores en el dominio de salida (mayor separación → más contraste), mientras que los niveles raros se comprimen.

El algoritmo completo, en cinco etapas, se presenta en la Tabla 3.1.

Tabla 3.1: Algoritmo de ecualización de histograma.

Etapa	Operación	Fórmula
1	**Histograma**	$h[k] \leftarrow$ número de píxeles con intensidad k, $k = 0 \ldots L-1$
2	**Probabilidad**	$p[k] \leftarrow h[k]/MN$
3	**CDF**	$\text{cdf}[k] \leftarrow \sum_{j=0}^{k} p[j]$ (suma acumulada)
4	***Look-Up Table* (mapeo)**	$\text{lut}[k] \leftarrow \text{round}(\text{cdf}[k] \times (L-1))$
5	**Aplicación**	$g[i,j] \leftarrow \text{lut}[f[i,j]]$ (para todo píxel)

Observe en la Figura 3.7 que la ecualización **redistribuye** los tonos existentes a posiciones más espaciadas en el rango $[0, L-1]$, pero no crea tonos nuevos — la imagen ecualizada continúa con exactamente 3 tonos distintos, ahora en $\{1, 5, 7\}$ en lugar de $\{2, 3, 4\}$.

```
img5 = np.array([[3, 4, 2, 3, 4],
                 [4, 3, 3, 4, 3],
                 [2, 3, 4, 3, 2],
                 [3, 4, 3, 2, 3],
                 [4, 3, 2, 3, 4]], dtype=np.uint8)
L = 8  # 3 bits: niveles 0..7

# 1. Histograma con tamaño garantizado hasta L
h = mm.hist(img5, 3)  # B=3 → 2³=8 niveles, tamaño garantizado

p   = h / h.sum()  # 2. Probabilidad de cada nivel p[k] = h[k] / MN

cdf = np.cumsum(p) # 3. CDF normalizada o
# 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: mapeo para [0, L-1]

img5_eq = lut[img5] # 5. Aplica LUT pixel a pixel o, equivalente a:
# l, c = img5.shape
# img5_eq = np.zeros((l, c), dtype=np.uint8)
# for i in range(l):
#     for j in range(c):
#         img5_eq[i, j] = lut[img5[i, j]]      # g[i,j] = lut[f[i,j]]

print("Imagen original (5×5, 3 bits):")
print(img5)
print()
header = f"{'k':>3} {'h[k]':>6} {'p[k]':>7} {'CDF[k]':>8} {'lut[k]':>7}"
print(header)
```

```python
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("Imagen ecualizada (5×5):")
print(img5_eq)

# Cuenta tonos distintos
tons_orig = len(np.unique(img5))
tons_eq   = len(np.unique(img5_eq))

mm.show(
    [img5, img5_eq, mm.histImg(img5, L-1), mm.histImg(img5_eq, L-1)],
    titles=[f"Original ({tons_orig} tonos: {sorted(np.unique(img5).tolist())})",
            f"Ecualizada ({tons_eq} tonos: {sorted(np.unique(img5_eq).tolist())})",
            "Histograma - Original",
            "Histograma - Ecualizada"],
    rows=2, cols=2, figsize=(5, 4), dpi=100
)
```

```
Imagen original (5×5, 3 bits):
[[3 4 2 3 4]
 [4 3 3 4 3]
 [2 3 4 3 2]
 [3 4 3 2 3]
 [4 3 2 3 4]]

  k   h[k]    p[k]   CDF[k]  lut[k]
-----------------------------------
  0      0  0.0000   0.0000       0
  1      0  0.0000   0.0000       0
  2      5  0.2000   0.2000       1
  3     12  0.4800   0.6800       5
  4      8  0.3200   1.0000       7
  5      0  0.0000   1.0000       7
  6      0  0.0000   1.0000       7
  7      0  0.0000   1.0000       7

Imagen ecualizada (5×5):
[[5 7 1 5 7]
 [7 5 5 7 5]
 [1 5 7 5 1]
 [5 7 5 1 5]
 [7 5 1 5 7]]
```

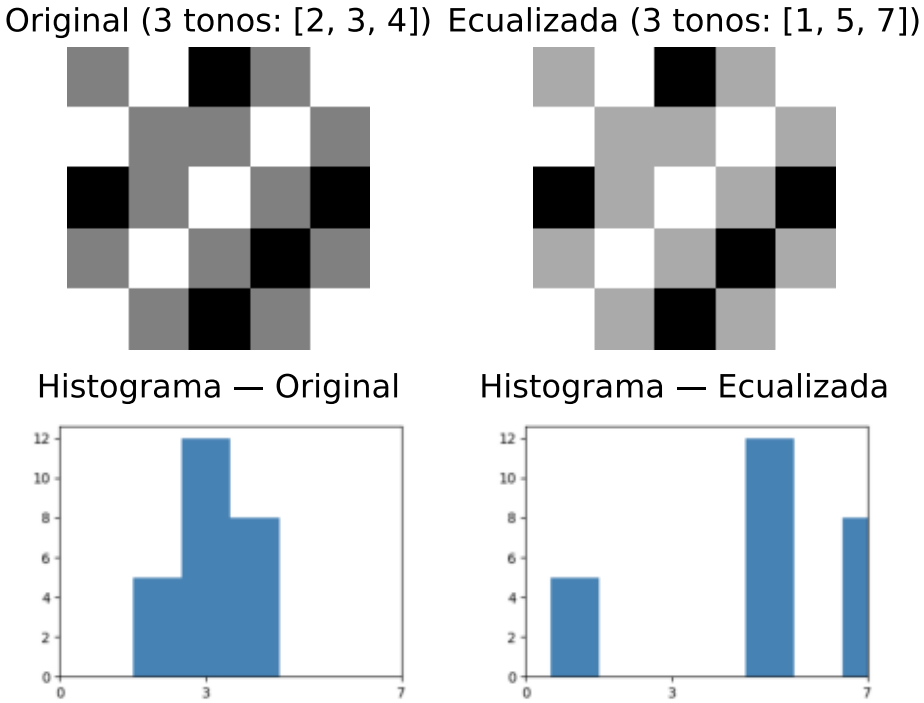


Figura 3.7: Ecualización de histograma en imagen 5×5 con 3 bits (L=8): imagen original con tonos concentrados en {2,3,4}, imagen ecualizada con tonos redistribuidos para {1,5,7}, y los respectivos histogramas evidenciando la dispersión de las frecuencias.

Limitación: la ecualización global puede realzar en exceso los ruidos y producir contraste excesivo en regiones homogéneas. El **CLAHE** (*Contrast Limited Adaptive Histogram Equalization*) reduce este problema al aplicar la ecualización en bloques locales (*tiles*) y limitar la altura de los picos del histograma antes de la ecualización.

La Figura 3.8 compara la imagen original, la ecualización global mediante `mm.equalize` y el CLAHE de OpenCV, mostrando también los histogramas resultantes. A diferencia de la ecualización global, que utiliza una única transformación basada en la CDF de toda la imagen, el CLAHE adapta el contraste a cada región, siendo particularmente útil en imágenes con iluminación no uniforme.

En el ejemplo, se utilizó `clipLimit=2.0` y `tileGridSize=(32,32)`. El parámetro `clipLimit` define cuánto pueden crecer los picos del histograma local antes de ser recortados (*clipped*). En OpenCV, este valor es un factor relativo: el límite real se calcula aproximadamente como `clipLimit × (número de píxeles del bloque / número de niveles de gris)`. Por ejemplo, en un bloque con 4096 píxeles y una imagen de 8 bits (256 niveles de gris), la frecuencia media por nivel es $4096/256 = 16$. Así, `clipLimit=2.0` permite picos de aproximadamente $2 \times 16 = 32$ ocurrencias antes del recorte. Las ocurrencias excedentes no se descartan: se redistribuyen entre los demás niveles de gris del histograma, reduciendo la concentración excesiva en pocos niveles y evitando una amplificación exagerada del contraste local. Valores menores limitan más el contraste y reducen la amplificación del ruido, mientras que valores mayores permiten un realce más intenso, pero pueden introducir artefactos.

- `clipLimit=1.0`: realce suave y conservador;
- `clipLimit=2.0`: buen equilibrio entre contraste y naturalidad;
- `clipLimit=4.0`: mayor énfasis en los detalles locales;
- `clipLimit=8.0`: contraste agresivo, con posible amplificación del ruido.

Así, el CLAHE suele producir resultados más naturales que la ecualización global, especialmente en imágenes con sombras, reflejos o iluminación desigual.

```
img_eq    = mm.equalize(img_gray)
clahe     = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(32, 32))
img_clahe = clahe.apply(img_gray)

images = [img_gray, img_eq, img_clahe]
titles = ["Original", "mm.equalize (CDF)", "CLAHE"]

mm.show(
    images + [mm.histImg(img) for img in images],
    titles=titles + [f"Histograma - {t}" for t in titles],
    rows=2, cols=3,
    figsize=(14, 8)
)
```

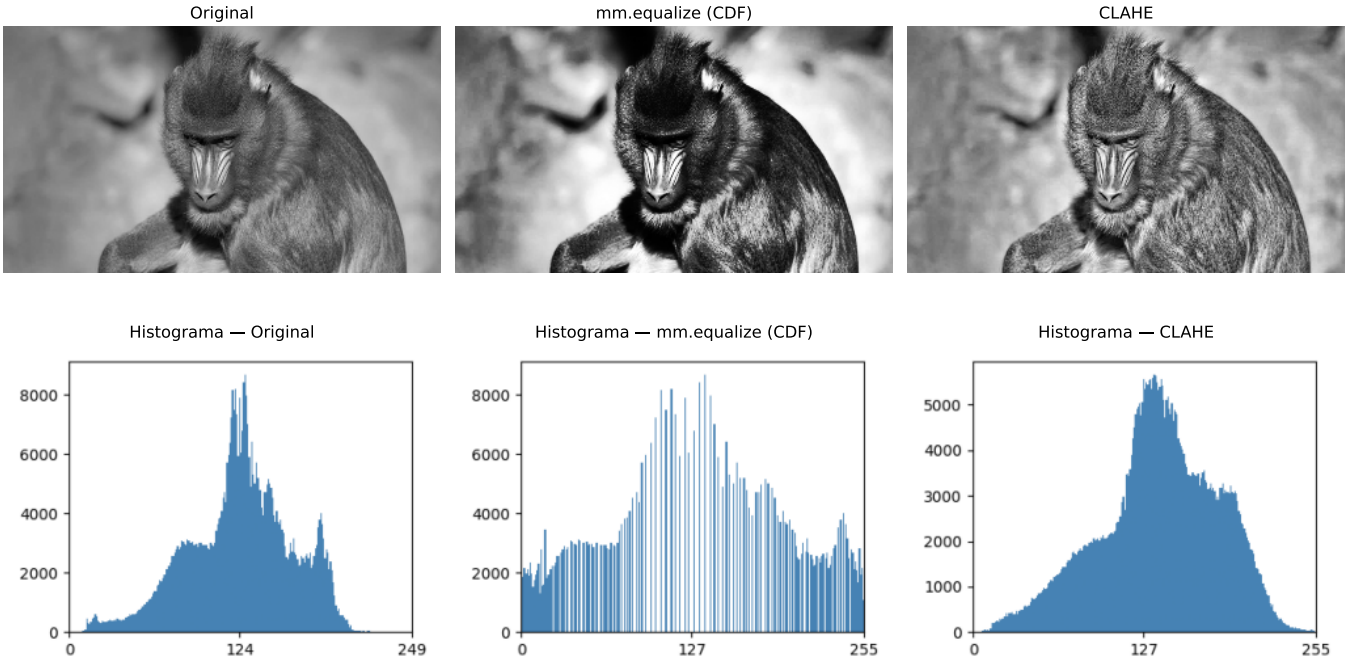


Figura 3.8: Ecualización de histograma: mm.equalize (global vía CDF) vs. CLAHE (adaptativa con limitación de contraste). Los histogramas revelan la dispersión progresiva de las intensidades.

3.3.2 Especificación de Histograma

Mientras que la ecualización impone una distribución uniforme, la **especificación de histograma** (*histogram matching*) permite que el histograma de la imagen de salida siga una distribución **arbitraria** — por ejemplo, el histograma de otra imagen de referencia.

El procedimiento involucra tres etapas:

1. Calcular la CDF de la imagen de entrada: $P_r(r_k)$.
2. Calcular la CDF de la imagen de referencia: $P_z(z_k)$.
3. Para cada nivel r_k, encontrar el nivel z que minimiza $|P_z(z) - P_r(r_k)|$.

$$T(r_k) = \arg\min_z |P_z(z) - P_r(r_k)| \tag{3.8}$$

En la Figura 3.9, transferimos el perfil tonal del leopardo (Figura 3.3) a la imagen del mandril — una aplicación directa del concepto visto en el *blending*: en lugar de fusionar píxeles, aquí fusionamos distribuciones tonales.

```python
# mapea src al perfil tonal de ref vía 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=["Mandril (original)", "Leopardo (referencia)", "Mandril → perfil del leopardo",
            "Histograma - Mandril", "Histograma - Leopardo", "Histograma - resultado"],
    rows=2, cols=3, figsize=(12, 9)
)
```

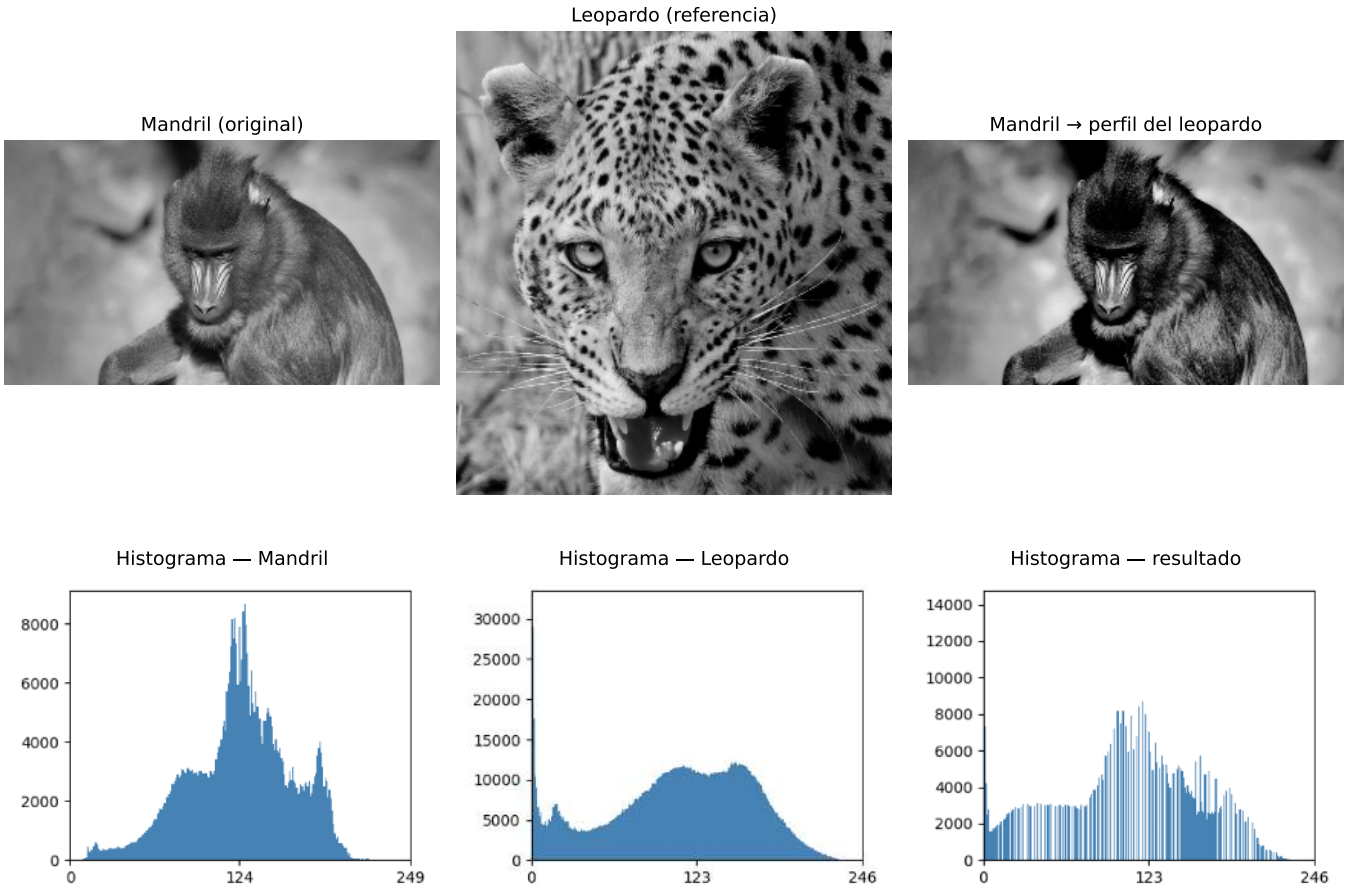


Figura 3.9: Especificación de histograma: mandril mapeada al perfil tonal del leopardo. La CDF de salida aproxima la CDF de referencia.

3.4 Fundamentos Espaciales: Vecindad, Convolución y *Kernels*

Las operaciones de filtrado espacial no actúan sobre un píxel aislado, sino sobre una **vecindad** que lo rodea. Para ello, se utiliza una pequeña matriz de coeficientes denominada **kernel** (o máscara), que recorre toda la imagen mediante una **ventana deslizante** (*sliding window*).

Las ventanas más comunes son de 3×3, 5×5 y 7×7. En una ventana de 3×3, por ejemplo, el píxel central se procesa junto con sus ocho vecinos inmediatos. En cada posición de la ventana, los valores de los píxeles se combinan con los coeficientes del *kernel*, produciendo un nuevo valor para el píxel central.

3.4.1 Vecindad

Considere una ventana de 3×3 centrada en el píxel (x, y):

$$\begin{bmatrix} (x-1, y-1) & (x, y-1) & (x+1, y-1) \\ (x-1, y) & (x, y) & (x+1, y) \\ (x-1, y+1) & (x, y+1) & (x+1, y+1) \end{bmatrix} \tag{3.9}$$

De forma general, una ventana de tamaño $(2a+1) \times (2b+1)$ abarca todos los píxeles situados hasta a posiciones en horizontal y hasta b posiciones en vertical con respecto al píxel central. Así, una ventana de 3×3 corresponde a $a = b = 1$, una ventana de 5×5 a $a = b = 2$, y así sucesivamente.

Matemáticamente, la vecindad se define como

$$\mathcal{V}(x, y) = \{(x+s,\, y+t) : -a \leq s \leq a,\, -b \leq t \leq b\} \tag{3.10}$$

3.4.2 Tratamiento de Bordes

Los píxeles cercanos a los bordes tienen parte de su vecindad fuera de la imagen. Para aplicar filtros en estas regiones, es necesario definir cómo se obtendrán los valores externos. Las tres estrategias más comunes (con la constante equivalente de OpenCV entre paréntesis) son:

- **Zero-padding** (`BORDER_CONSTANT`): completa la región externa con ceros.
- **Replicación** (`BORDER_REPLICATE`): repite el valor del píxel del borde.
- **Reflexión** (`BORDER_REFLECT_101`): refleja los píxeles vecinos, sin repetir el del borde.

`mm.conv` usa la reflexión por defecto, ya que preserva mejor la continuidad de los niveles de gris y reduce artefactos en el tratamiento de los bordes.

El siguiente ejemplo compara las tres estrategias con `mm.pad` en una matriz 3×3. Observa cómo cada una rellena los píxeles externos necesarios para aplicar un filtro 3×3 también en las esquinas.

```
img = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
], dtype=np.uint8)

bordas = ["constant", "replicate", "reflect101"]

print("Imagen original:")
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)))
```

```
Imagen original:
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
```

Observa que el resultado de un filtro puede variar significativamente según el tratamiento adoptado para los bordes de la imagen.

En la biblioteca didáctica `morph.py`, los algoritmos implementados directamente en Python (sin OpenCV ni *scikit-image*) — las variantes de sufijo `0`, como `mm.conv0` — no usan *padding*. Para cada píxel, los elementos de la vecindad se examinan individualmente, y solo los vecinos cuyas coordenadas pertenecen al dominio de la imagen participan en el cálculo. Así, en las regiones próximas a los bordes, la vecindad efectiva puede contener menos píxeles que la ventana especificada; en los operadores basados en `mm._viz` (por ejemplo, `mm.laplacian`), los píxeles de borde con ventana incompleta se dejan inalterados.

3.4.3 Correlación vs. Convolución

Existen dos mecanismos matemáticamente relacionados.

Correlación cruzada (*cross-correlation*) — el *kernel* se aplica directamente:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)\, f(x+s,\, y+t) \tag{3.11}$$

Convolución bidimensional — el *kernel* se rota 180° antes de su aplicación:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)\, f(x-s,\, y-t) \tag{3.12}$$

Para *kernels* simétricos (Gaussiano, Laplaciano, media) las dos operaciones producen resultados idénticos. Para *kernels* asimétricos (Sobel, Prewitt) la diferencia es significativa, como lo muestran los ejemplos a continuación.

3.4.3.1 Correlación (`mm.conv`)

```
# imagen 4x4 (uint8) y kernel asimétrico 3x3
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")  # cero fuera de la imagen

print("Imagen original:");         print(mm.drawImg(img))
print("Kernel:");                  print(mm.drawImg(w))
print("Resultado de la correlación:"); print(mm.drawImg(corr))
```

```
Imagen original:
 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

Resultado de la correlación:
 0  0  0  0
 5  8 11  4
17 20 23  8
29 32 35 12
```

`mm.conv` realiza correlación, es decir, aplica el *kernel* exactamente en la orientación proporcionada.

3.4.3.2 Convolución

```
# repetidos aquí para que la celda sea independiente
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)

# núcleo rotado 180° (equivale a 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("Imagen original:");            print(mm.drawImg(img))
print("Núcleo original:");            print(mm.drawImg(w))
print("Núcleo rotado 180°:");    print(mm.drawImg(w_conv))
print("Resultado de la convolución:");    print(mm.drawImg(conv))
```

```
Imagen original:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16

Núcleo original:
  0   1   2
  0   0   0
  0   0   0

Núcleo rotado 180°:
  0   0   0
  0   0   0
  2   1   0

Resultado de la convolución:
 5 16 19 22
 9 28 31 34
13 40 43 46
 0  0  0  0
```

La convolución utiliza el *kernel* rotado 180°. Para reproducir la definición matemática de convolución, se rota el *kernel* (aquí, `[[0,1,2],[0,0,0],[0,0,0]]` → `[[0,0,0],[0,0,0],[2,1,0]]`) antes de aplicar `mm.conv`.

3.4.4 El Papel del *Kernel*

Los coeficientes del *kernel* determinan completamente el efecto producido por el filtro, tal como se resume en la Tabla 3.2.

Tabla 3.2: Interpretación típica de los coeficientes del *kernel*.

Característica	Efecto típico
Coeficientes positivos con suma 1	Suavizado (*pasa-baja*)
Suma igual a 0, con valores positivos y negativos	Detección de bordes (*pasa-alta*)
Coeficiente central positivo dominante y vecinos negativos	Realce de nitidez
Coeficientes asimétricos	Gradiente direccional

Ejemplos:

Suavizado:

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

Detección de bordes:

$$\begin{bmatrix}-1 & -1 & -1\\ -1 & 8 & -1\\ -1 & -1 & -1\end{bmatrix}$$

Realce de nitidez:

$$\begin{bmatrix}0 & -1 & 0\\ -1 & 5 & -1\\ 0 & -1 & 0\end{bmatrix}$$

Gradiente direccional (Sobel):

$$\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix}$$

La Figura 3.10 demuestra el mecanismo paso a paso: para cada posición de la ventana, se multiplica cada coeficiente del *kernel* por el píxel correspondiente de la vecindad y se suman los productos obtenidos. El resultado es exactamente el valor definido por la Ecuación 3.11 para esa posición de la imagen. Aunque las imágenes producidas por `mm.conv0` y `cv2.filter2D` (o `mm.conv`) sean visualmente muy similares, la implementación basada en OpenCV es miles de veces más rápida, como se muestra a continuación.

Rendimiento: bucles de Python vs. operaciones vectorizadas

La función `mm.conv0` implementa la correlación directamente en Python mediante bucles anidados. Aunque este enfoque es adecuado para fines didácticos, ejecuta un gran número de operaciones y se vuelve lento para imágenes más grandes.
En cambio, `mm.conv` utiliza `cv2.filter2D`, implementado en C++ y optimizado para operaciones matriciales. En el ejemplo presentado, la versión vectorizada fue más de **3000 veces más rápida** que la implementación didáctica, produciendo un resultado visualmente equivalente.
Las diferencias numéricas observadas se concentran principalmente en los bordes de la imagen. En `mm.conv0`, los píxeles del borde permanecen inalterados, mientras que `mm.conv` utiliza una estrategia de reflexión de bordes (`cv2.BORDER_REFLECT_101`, estándar de `cv2.filter2D`).
Por ello, `mm.conv0` debe utilizarse para comprender el algoritmo, mientras que `mm.conv` es la opción recomendada para aplicaciones prácticas.

```
import time

w_mean = np.ones((3,3), dtype=np.float32) / 9.0
img_gray = img_leop_gray
# Demostración numérica en parche 5×5
patch = img_gray[250:255, 250:255].astype(np.float32)
roi   = patch[0:3, 0:3]
print("Parche 5×5 (intensidades):")
print(patch.astype(np.int32))
print(f"\nKernel 3×3 de media:\n{w_mean}")
print(f"\nCorrelación en el píxel central [1,1]: \
      {(w_mean * roi).sum():.1f}  (original: {patch[1,1]:.0f})")

# Comparación de tiempo en la imagen completa
t0 = time.perf_counter()
img_conv0 = mm.conv0(img_gray, w_mean)
t_conv0 = time.perf_counter() - t0
```

```python
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"\nTiempos en la imagen {img_gray.shape[0]}x{img_gray.shape[1]}:")
print(f"  mm.conv0 (lazos Python): {t_conv0:.3f} s")
print(f"  mm.conv  (cv2.filter2D): {t_conv:.5f} s")
print(f"  Aceleración:              {t_conv0/t_conv:.0f}× más rápido")
print(f"  Diferencia máxima:        {diff} (bordes con padding diferente)")

mm.show(
    [img_gray, img_conv0, img_conv],
    titles=["Original", f"conv0 ({t_conv0:.2f}s)", f"conv ({t_conv:.4f}s)"],
    cols=3
)
```

```
Parche 5×5 (intensidades):
[[ 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]]

Kernel 3×3 de media:
[[0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]]

Correlación en el píxel central [1,1]:       103.0  (original: 107)

Tiempos en la imagen 1324x1242:
  mm.conv0 (lazos Python): 6.192 s
  mm.conv  (cv2.filter2D): 0.00313 s
  Aceleración:              1978× más rápido
  Diferencia máxima:        39 (bordes con padding diferente)
```

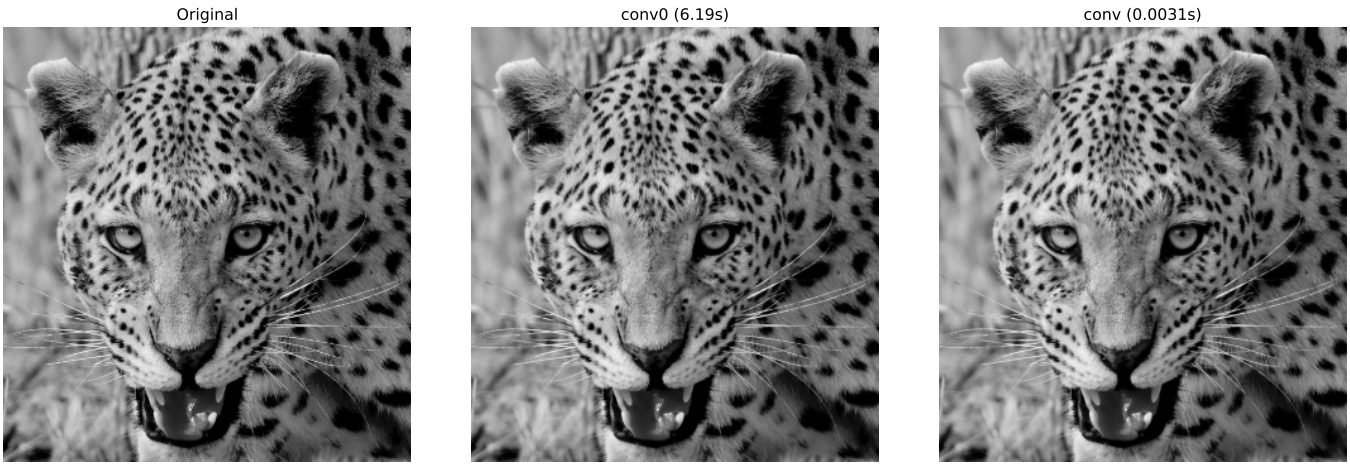


Figura 3.10: Correlación con *kernel* de media 3×3: versión didáctica (mm.conv0) vs. vectorizada (mm.conv vía cv2.filter2D). La diferencia máxima de 39 ocurre en los bordes.

3.4.5 Ejemplo Numérico: Correlación Paso a Paso

Para hacer concreto el mecanismo de la Ecuación 3.11, considere el *kernel* de media 3×3 ($a = b = 1$, todos los coeficientes $= 1/9 \approx 0{,}111$) aplicado al *patch* 5×5 extraído de la imagen del leopardo. La Figura 3.11 muestra el *patch* con la cuadrícula y resalta en amarillo la ventana 3×3 centrada en el píxel $[1, 1]$:

```
patch = img_gray[250:255, 250:255]
B     = np.ones((3, 3), dtype=np.uint8)

print("Patch 5×5 (intensidades):")
print(mm.drawImg(patch))

mm.drawImgKernel(patch, B, 1, 1, 40)
```

```
Patch 5×5 (intensidades):
 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
```

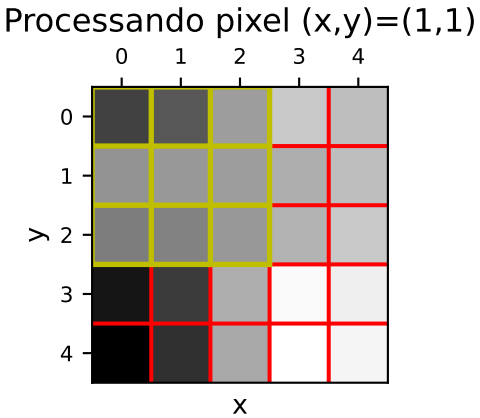


Figura 3.11: *Patch* 5×5 extraído de la imagen del leopardo (posición [250:255, 250:255]). La ventana amarilla resalta la vecindad 3×3 centrada en el píxel [1,1] donde se calculará la correlación.

Para ilustrar el cálculo de la correlación, considere el píxel en la posición $[1, 1]$ del *patch* 5×5 mostrado en la Figura 3.11.. Esa posición fue elegida solo por conveniencia didáctica, ya que posee una vecindad 3×3 completa a su alrededor.

Los valores de esa vecindad corresponden a la submatriz superior izquierda del *patch*:

$$\text{vecindad} = \begin{bmatrix} 91 & 95 & 108 \\ 106 & 107 & 108 \\ 102 & 103 & 107 \end{bmatrix}$$

Aplicando la Ecuación 3.11 con el kernel de media:

$$g[1, 1] = \frac{91 + 95 + 108 + 106 + 107 + 108 + 102 + 103 + 107}{9} = \frac{927}{9} = 103$$

El resultado (103) es ligeramente menor que el valor original del píxel central (107), pues la media incorpora vecinos de menor intensidad, produciendo el efecto de suavizado. En la práctica, el algoritmo inicia el procesamiento en $[0, 0]$ y repite ese mismo cálculo para cada posición de la imagen, desplazando la ventana hasta cubrir todo el dominio.

3.5 Filtrado Espacial de Suavizado

Los filtros de suavizado (*smoothing filters*) atenúan variaciones bruscas de intensidad, reduciendo ruido y detalles de alta frecuencia. Son filtros **paso-bajo** — preservan las componentes de baja frecuencia (estructuras grandes) y atenúan las de alta frecuencia (ruido, bordes).

3.5.1 Filtro de Media (*Box Filter*)

El filtro de media utiliza un *kernel* uniforme de tamaño $n \times n$, donde todos los coeficientes valen $1/n^2$:

$$w_{\text{media}} = \frac{1}{n^2} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}_{n \times n} \tag{3.13}$$

Cada píxel de salida es la media aritmética de los n^2 píxeles de su vecindario. Nótese que la suma de los coeficientes es siempre 1 — el brillo medio de la imagen se preserva. Los *kernels* más grandes producen un suavizado más agresivo, pero desenfocan progresivamente los bordes.

Figura 3.12 muestra el efecto del filtro de media con *kernels* 3×3, 7×7 y 15×15 sobre un detalle de la imagen del leopardo. Los resultados se obtuvieron con `mm.blur`, que implementa el filtro de media mediante la función `cv2.blur`, equivalente a la convolución de la imagen con un *kernel* uniforme cuyos coeficientes son $h(x, y) = 1/N^2$; de forma equivalente, el mismo resultado puede obtenerse con `mm.conv`, calculando ($g = f * h$). A medida que el *kernel* aumenta, más píxeles contribuyen a cada valor de salida, intensificando el suavizado, reduciendo el ruido y haciendo que los detalles finos y los bordes se vuelvan progresivamente más borrosos.

```
# Detalle de la región del ojo
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] # o
    #[mm.conv(img_gray_crop, np.ones((k,k), dtype=np.float32)/(k*k)) for k in sizes]
titles = ["Original"] + [f"Média {k}×{k}" for k in sizes]

mm.show(imgs, titles=titles, cols=4)
```

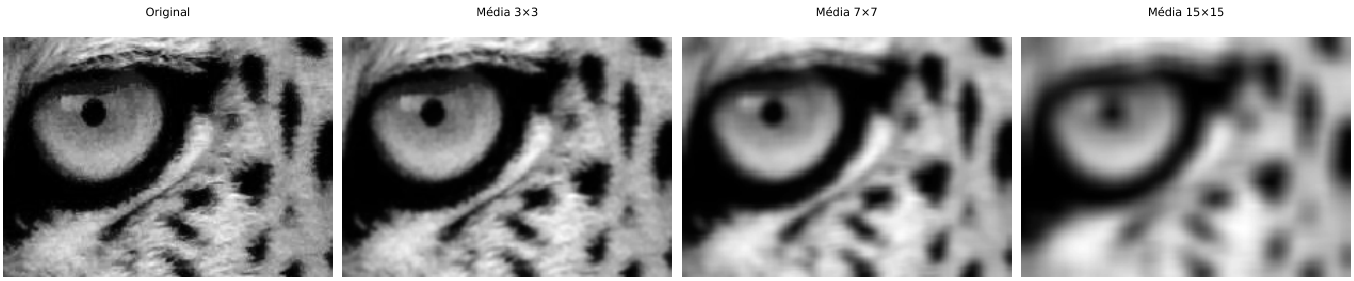


Figura 3.12: Filtro de media con *kernels* de tamaño creciente (3×3, 7×7, 15×15). El desenfoque de los bordes aumenta con el tamaño del *kernel*.

3.5.2 Filtro Gaussiano

El filtro Gaussiano pondera los píxeles de la vecindad de acuerdo con una función Gaussiana bidimensional:

$$G(s,t) = \frac{1}{2\pi\sigma^2}\, e^{-\frac{s^2+t^2}{2\sigma^2}} \tag{3.14}$$

donde σ es la desviación estándar y controla el radio de influencia. Los píxeles más cercanos al centro tienen un peso mayor; los píxeles distantes se ignoran progresivamente.

La Figura 3.13 presenta el *kernel* Gaussiano 5×5 generado para $\sigma = 1$. El *kernel* fue construido a partir del producto externo de dos vectores Gaussianos unidimensionales y posteriormente normalizado para que la suma de sus coeficientes sea igual a 1. Se observa que los mayores pesos se concentran en el centro de la matriz, decreciendo radialmente hacia los bordes. Esta distribución hace que los píxeles centrales tengan mayor influencia en el resultado del filtrado, contribuyendo a una suavización más natural y con mejor preservación de bordes que el filtro de media.

```
# Genera núcleo Gaussiano 5×5 vía cv2
k_gauss = cv2.getGaussianKernel(5, 1)
w_gauss = k_gauss @ k_gauss.T          # @ => multiplicación matricial: G(s,t) = G(s)·G(t)
w_gauss = w_gauss / w_gauss.sum()      # garantiza suma = 1

print("Núcleo Gaussiano 5×5 ( =1), normalizado:")
for row in w_gauss:
    print("  " + "  ".join(f"{v:.4f}" for v in row))
print(f"\nSuma de los coeficientes: {w_gauss.sum():.6f}")
print(f"Peso central vs. esquina: {w_gauss[2,2]:.4f} vs. {w_gauss[0,0]:.4f} "
      f"({w_gauss[2,2]/w_gauss[0,0]:.1f}× mayor)")

mm.drawImgPlt((w_gauss * 1000).astype(np.uint8), scale=40)
```

```
Núcleo Gaussiano 5×5 ( =1), normalizado:
  0.0030  0.0133  0.0219  0.0133  0.0030
  0.0133  0.0596  0.0983  0.0596  0.0133
  0.0219  0.0983  0.1621  0.0983  0.0219
  0.0133  0.0596  0.0983  0.0596  0.0133
  0.0030  0.0133  0.0219  0.0133  0.0030

Suma de los coeficientes: 1.000000
Peso central vs. esquina: 0.1621 vs. 0.0030 (54.6× mayor)
```

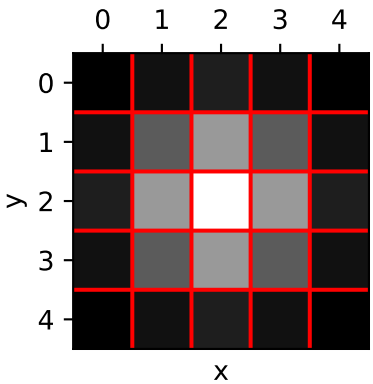


Figura 3.13: *Núcleo* Gaussiano 5×5 (=1): pesos normalizados, mayores en el centro y decrecientes radialmente. La cuadrícula facilita la lectura de cada coeficiente.

Ventaja computacional de la separabilidad

Considere un *kernel* cuadrado de tamaño $n \times n$. Si este filtro es separable (como el Gaussiano), la convolución 2D puede descomponerse en dos convoluciones 1D: una horizontal y otra vertical.
En ese caso, el costo por píxel pasa de aproximadamente $O(n^2)$ operaciones (convolución 2D directa) a $O(2n)$ operaciones (dos convoluciones 1D). Así, la complejidad se reduce de forma significativa, haciendo el procesamiento más eficiente.

En comparación con el filtro de media, el Gaussiano:

- **Preserva mejor los bordes** — la ponderación radial suaviza sin crear transiciones abruptas;
- **No introduce anillos** (*ringing*) en el dominio de la frecuencia, pues la Gaussiana es su propia transformada de Fourier (Capítulo 5);
- **Está controlado por** σ — aumentar σ equivale a aumentar el radio de suavizado de forma continua y predecible.

La Figura 3.14 compara los filtros de media y gaussiano aplicados a la imagen del leopardo usando una ventana 9×9. El filtro de media se implementó mediante convolución con un *kernel* uniforme, donde todos los 81 píxeles de la vecindad poseen el mismo peso (1/81), mientras que el filtro gaussiano se obtuvo con `cv2.GaussianBlur`, utilizando pesos definidos por una distribución gaussiana. Ambos reducen ruido y suavizan la imagen, pero el filtro gaussiano preserva mejor los bordes y los detalles locales, como puede observarse en la región ampliada del ojo.

La Figura 3.14 compara los filtros de media y gaussiano aplicados a un detalle de la imagen del leopardo con *kernels* 9×9. El filtro de media se obtuvo con `mm.blur`, equivalente a la convolución con un *kernel* uniforme cuyos coeficientes valen 1/81, mientras que el filtro gaussiano se obtuvo con `mm.gaussian`, equivalente a la convolución con un *kernel* generado a partir de una distribución gaussiana. Ambos promueven suavización y reducción de ruido, pero el filtro gaussiano asigna mayor peso a los píxeles centrales de la vecindad, preservando mejor los bordes y los detalles locales, como puede observarse en la región ampliada del ojo.

```
img_media9 = mm.blur(img_gray, 9)
img_gauss9 = mm.gaussian(img_gray, 9, 0)

# Detalle de la región del ojo
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=["Detalle: Original", "Media 9×9", "Gaussiano 9×9"],
    cols=3, figsize=(12, 8)
)
```

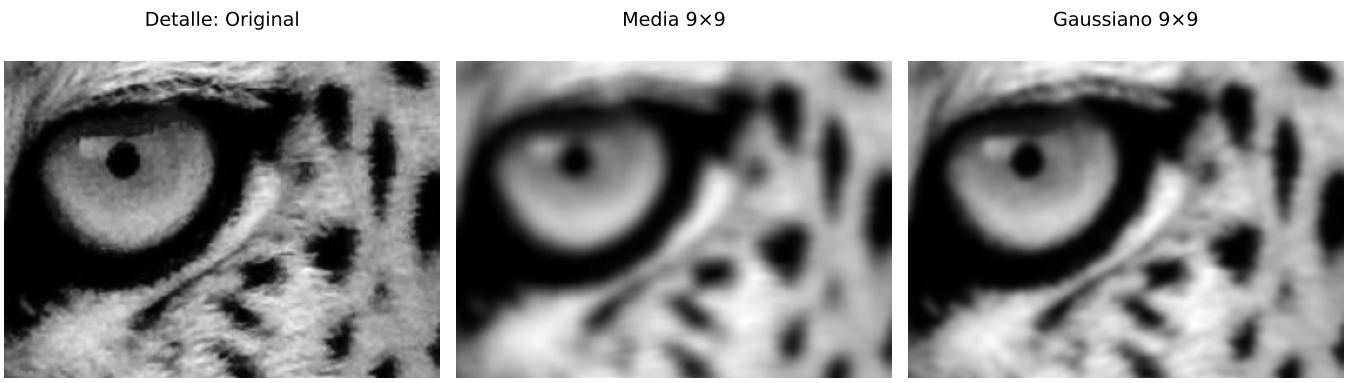


Figura 3.14: Comparación entre filtro de media y Gaussiano (*kernel* 9×9, =0). El Gaussiano preserva mejor los bordes, visible en el detalle del rostro.

3.6 Filtrado Espacial de Realce

Los filtros de realce (*sharpening filters*) enfatizan transiciones abruptas de intensidad, aumentando la nitidez y la visibilidad de bordes. Son filtros **paso-alto** — amplifican los componentes de alta frecuencia (bordes, textura) y suprimen los de baja frecuencia (regiones uniformes).

La intuición es simple: si restamos de una imagen su versión suavizada (que contiene solo las bajas frecuencias), lo que queda son las altas frecuencias — bordes y detalles. Sumando ese residuo de vuelta a la imagen original, el contraste local aumenta:

$$g = f + k\,(f - f_{\text{suave}}), \quad k > 0 \tag{3.15}$$

La Figura 3.15 ilustra este proceso en una señal 1D sintética con tres estructuras distintas: un escalón ancho, un pico fino y una rampa suave. En el panel , la señal original $f(x)$; en el , la versión suavizada $f_{\text{suave}}(x)$ obtenida por media móvil — nótese cómo el pico fino se atenúa. El panel muestra el residuo $f - f_{\text{suave}}$, que retiene solo las transiciones abruptas. Finalmente, el panel muestra $g(x)$: el pico, antes atenuado, se restaura y amplifica en relación con el original. Ajuste k y el tamaño de la ventana para observar el *trade-off* entre nitidez y amplificación de ruido.

Los filtros de realce formalizan esta idea directamente en el *kernel*, sin necesidad de dos etapas separadas.

3.6.1 Laplaciano

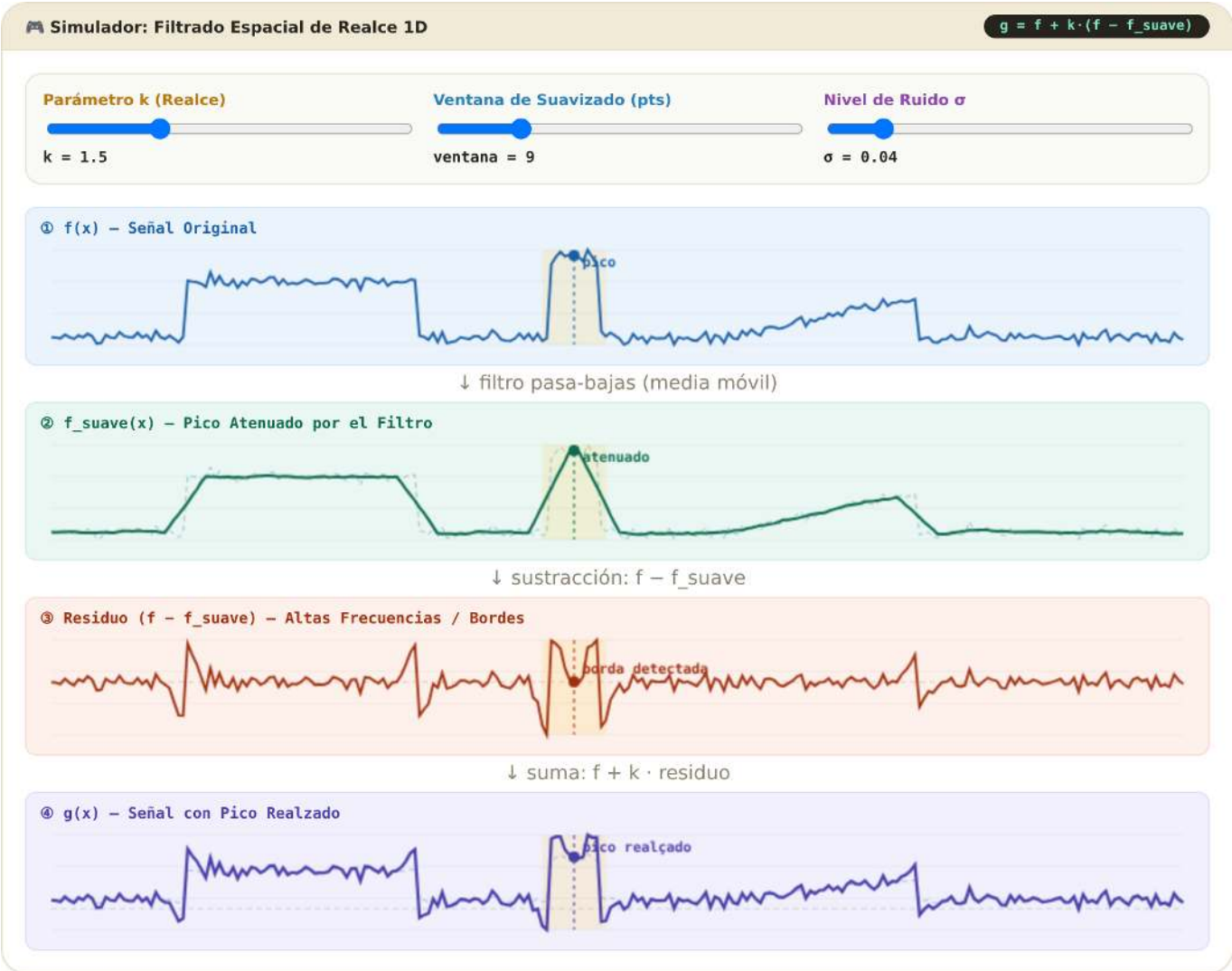
El Laplaciano es un operador de **segunda derivada** isotrópico, es decir, responde de igual manera a las variaciones en todas las direcciones, a diferencia de los operadores de primera derivada, como Sobel y Prewitt, que son direccionales:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{3.16}$$

Una propiedad importante de la segunda derivada es que su valor es **cercano a cero en regiones uniformes** y elevado en las transiciones de intensidad. Así, al restar el Laplaciano de la imagen original, se refuerzan los bordes y detalles, aumentando el contraste local:

$$g(x, y) = f(x, y) - \nabla^2 f(x, y) \tag{3.17}$$

En la forma discreta, la segunda derivada en x se aproxima por $f(x+1, y) - 2f(x, y) + f(x-1, y)$, y de manera análoga en y. Sumando ambas direcciones, se obtiene el *kernel* w_4 (4-vecinos) o w_8 (8-vecinos, incluyendo diagonales):



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap03/sim-03-filtragem1d.html

Figura 3.15: Simulador: Filtrado Espacial de Realce 1D (Unsharp Masking y High-Boost)

$$w_4 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \qquad w_8 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.18}$$

Suma cero y centro negativo

Ambos *kernels* tienen **suma de coeficientes igual a cero**: en regiones uniformes, la salida es 0 — el Laplaciano no altera el brillo medio, solo detecta variaciones. El **centro negativo** indica que el píxel se compara con sus vecinos: cuanto más se destaque (hacia arriba o hacia abajo), mayor será el valor absoluto del Laplaciano en ese punto.

En el siguiente ejemplo, el píxel central $[1, 1] = 107$ posee vecinos $\{95, 106, 108, 103\}$. Como estos valores son cercanos entre sí, la región es casi uniforme y el Laplaciano devuelve un valor bajo, produciendo poco realce. En regiones de borde, donde hay diferencias mayores entre el píxel central y sus vecinos, el Laplaciano asume valores más elevados (positivos o negativos), y la operación de Ecuación 3.17 intensifica esas transiciones.

La Figura 3.16 ilustra el cálculo del Laplaciano con el *kernel* w_4 en una vecindad 3×3 destacada dentro de un *patch* 5×5. El ejemplo muestra el valor obtenido por el operador y el correspondiente píxel realzado en la imagen de salida, que pasa de 107 a 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("Parche original:")
print(mm.drawImg(patch))
print(f"Laplac. w4 en  [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"Píxel resaltado [1,1]: {p[1,1]:.0f} - ({lap_11}) = {real_11}")

mm.drawImgKernel(patch, B, x=1, y=1, scale=40.0)
```

```
Parche original:
 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 en  [1,1]: 95 + 106 - 4×107 + 108 + 103 = -16
Píxel resaltado [1,1]: 107 - (-16) = 123
```

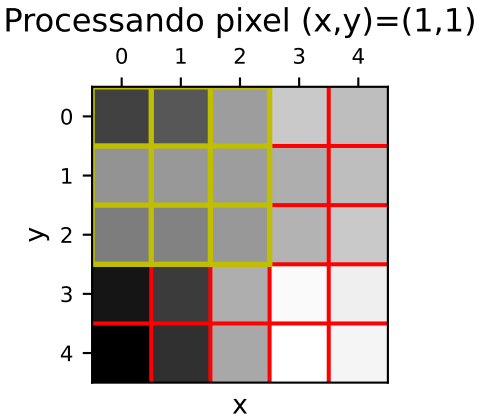


Figura 3.16: *Kernel* Laplaciano w4 aplicado al *patch* 5×5: la ventana amarilla resalta la vecindad 3×3 donde se calcula el operador de segunda derivada.

A Figura 3.17 compara a aplicação de los *kernels* laplacianos w_4 y w_8 en un recorte más grande de la imagen del leopardo. Para cada caso, se muestran la respuesta bruta del operador, que evidencia los bordes y las transiciones de intensidad, y la imagen obtenida tras el realce por sustracción del laplaciano. Se observa que el *kernel* w_8, al considerar también los vecinos diagonales, produce una respuesta más intensa y detecta variaciones en más direcciones, resultando en un realce ligeramente más acentuado.

```
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", "Laplaciano w4", "Realce w4",
            "Original", "Laplaciano w8", "Realce w8"],
    cols=3, figsize=(14, 8)
)
```

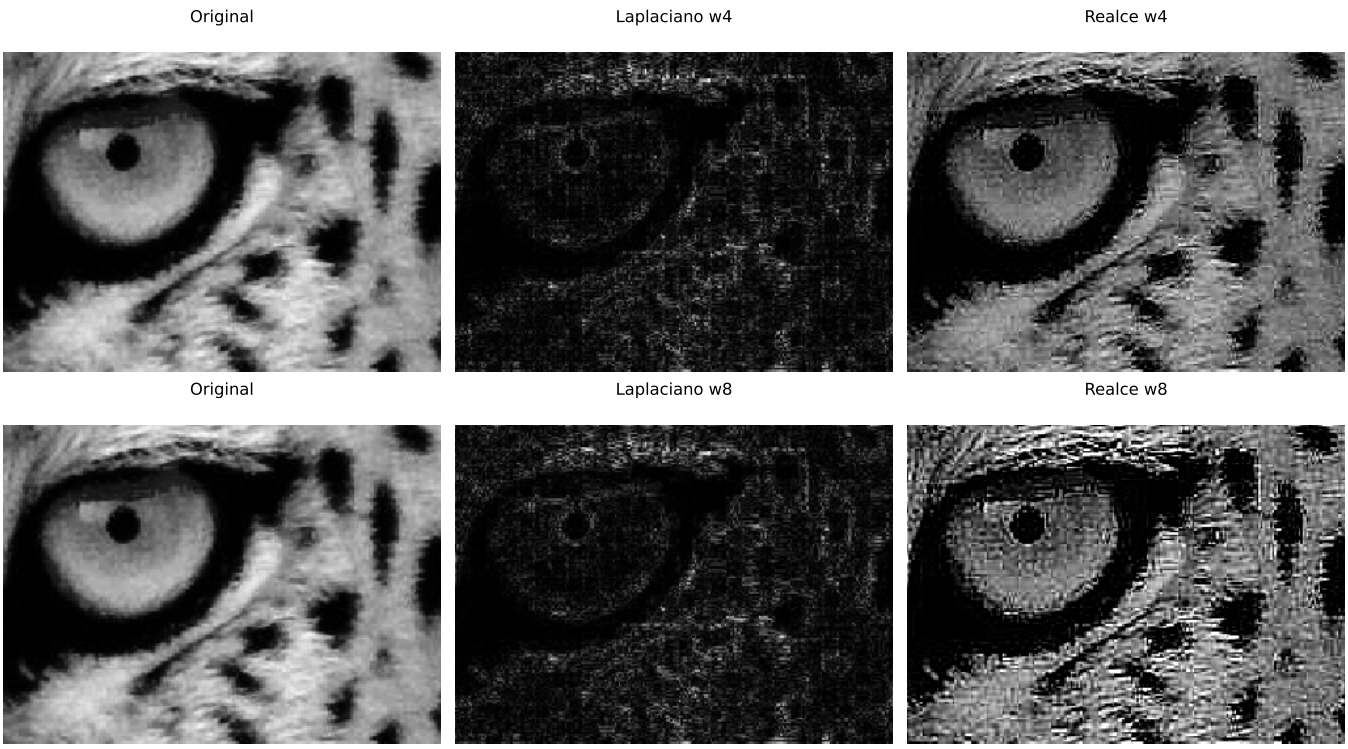


Figura 3.17: Laplaciano aplicado a la imagen del leopardo: respuesta cruda (bordes) con w4 y w8, e imágenes realzadas por la sustracción del Laplaciano. w8 es más sensible a las diagonales.

3.6.2 Operador de Sobel

El operador de Sobel estima las **derivadas parciales de primer orden** en las direcciones horizontal y vertical. A diferencia del Laplaciano (segunda derivada), el Sobel es direccional y más robusto al ruido, pues cada *kernel* combina una derivada con un suavizado Gaussiano perpendicular:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * f, \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * f \tag{3.19}$$

G_x detecta bordes **verticales** (variación en la dirección x); G_y detecta bordes **horizontales** (variación en la dirección y). Los pesos $\{1, 2, 1\}$ en la dirección perpendicular corresponden al suavizado Gaussiano 1D, que reduce la sensibilidad al ruido.

> **Sobel es correlación, no convolución**
>
> Los *kernels* de Sobel son **asimétricos** — la rotación de 180° altera el resultado. `cv2.Sobel` implementa correlación cruzada (como `cv2.filter2D`). Para obtener la derivada direccional correcta, las señales ya están definidas para correlación: G_x devuelve valores positivos donde la intensidad crece de izquierda a derecha.

La magnitud del **gradiente** combina los dos componentes, representando la fuerza del borde independientemente de la dirección:

$$|\nabla f| = \sqrt{G_x^2 + G_y^2} \tag{3.20}$$

Y la **dirección** del gradiente (perpendicular al borde) es:

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \tag{3.21}$$

Para ilustrar numéricamente, se calculan G_x y G_y manualmente en el píxel central $[1,1]$ del *patch* de 5×5:

```
patch = img_gray[250:255, 250:255]
p     = patch.astype(np.float32)

# Gx en el pixel [1,1]: columna derecha - columna izquierda (ponderada)
gx = (p[0,2] + 2*p[1,2] + p[2,2]) - (p[0,0] + 2*p[1,0] + p[2,0])

# Gy en el pixel [1,1]: fila inferior - fila superior (ponderada)
gy = (p[2,0] + 2*p[2,1] + p[2,2]) - (p[0,0] + 2*p[0,1] + p[0,2])

mag   = np.sqrt(gx**2 + gy**2)
theta = np.degrees(np.arctan2(gy, gx))

print("Parche 5x5:")
print(mm.drawImg(patch))
print(f"Gx en [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 en [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}")

# Sustituido el | por el prefijo mag para no romper el intérprete de tablas del Quarto
print(f"mag(Grad f)  = sqrt({gx:.1f}^2 + {gy:.1f}^2) = {mag:.1f}")
print(f"Theta        = arctan({gy:.1f}/{gx:.1f}) = {theta:.1f} grados")
```

```
Parche 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 en [1,1]: (108 + 2*108 + 107)       - (91 + 2*106 + 102) = 26.0
Gy en [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 grados
```

El valor reducido de $|\nabla f|$ en ese *patch* confirma que la región es casi uniforme, pues el gradiente asume valores elevados solo donde hay cambios significativos de intensidad. La Figura 3.18 aplica el operador de Sobel a un recorte mayor de la imagen del leopardo. Se presentan las respuestas horizontal (G_x) y vertical (G_y), obtenidas por convolución con los respectivos *kernels* de Sobel, además de la magnitud $|\nabla f|$, calculada a partir de la combinación de ambas. Mientras que G_x destaca bordes verticales y G_y bordes horizontales, la magnitud evidencia bordes en cualquier dirección.

```
def norm255(img):
    mn, mx = img.min(), img.max() #+1e-9 para evitar división por cero
    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))

# magnitud = mm.sobel(img_gray_crop) # o, pero con clip en lugar de norm255

mm.show(
```

```
    [img_gray_crop, norm255(np.abs(sobelx)), norm255(np.abs(sobely)),
     norm255(magnitude), norm255(direcao % 180)],
    titles=["Original", "Gx (bordes vert.)", "Gy (bordes horiz.)",
            "Magnitud | f|", "Dirección  (mod 180°)"],
    cols=5
)
```

Figura 3.18: Operador de Sobel en la imagen del leopardo: Gx detecta bordes verticales, Gy detecta bordes horizontales, y la magnitud | f| combina ambos, revelando todos los bordes independientemente de la dirección.

3.6.3 Operador de Prewitt

El operador de Prewitt es estructuralmente idéntico al de Sobel, pero sustituye la ponderación gaussiana $\{1, 2, 1\}$ por pesos uniformes $\{1, 1, 1\}$:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} * f, \qquad G_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} * f \tag{3.22}$$

La magnitud y la dirección del gradiente siguen las mismas ecuaciones que las de Sobel (Ecuación 3.20 y Ecuación 3.21). La diferencia práctica es que Prewitt es ligeramente más sensible al ruido — el suavizado perpendicular uniforme pondera menos el píxel central de la línea — pero computacionalmente más simple. En imágenes con bajo ruido los resultados son equivalentes.

```
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", "Magnitud | f|"],
    cols=4
)
```

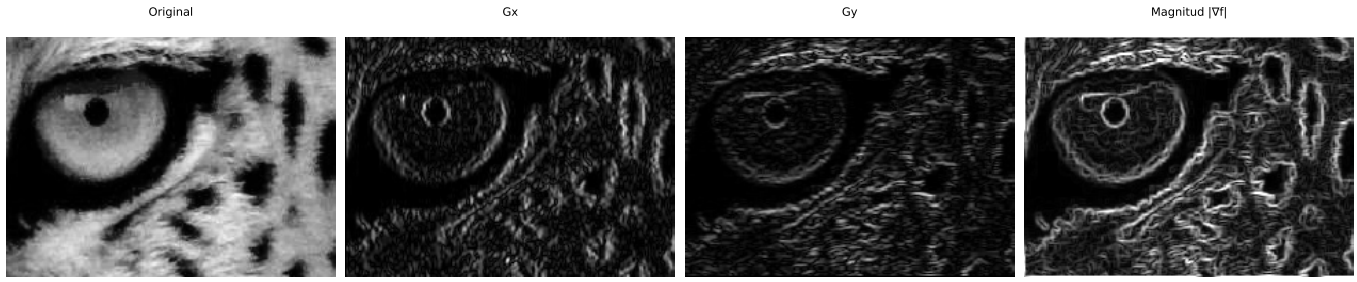


Figura 3.19: Operador de Prewitt: Gx, Gy y magnitud — comparable al Sobel, pero sin ponderación Gaussiana perpendicular.

3.6.4 *Unsharp Masking* (USM)

El *Unsharp Masking* es una técnica clásica de realce de nitidez originaria de la fotografía analógica, hoy ampliamente utilizada en software de edición de imágenes. La idea central es extraer los **componentes de alta frecuencia** de la imagen (bordes y detalles) y sumarlos de vuelta a la original con un peso k:

Tabla 3.3: Etapas del *Unsharp Masking*.

Etapa	Operación	Descripción
1	$\bar{f} = f * G_\sigma$	Suaviza con gaussiana — retiene bajas frecuencias
2	$m = f - \bar{f}$	Máscara: diferencia = altas frecuencias (bordes)
3	$g = f + k \cdot m$	Suma ponderada de la máscara a la original

Sustituyendo la etapa 2 en la etapa 3, se obtiene la expresión compacta:

$$g = f + k\,(f - f * G_\sigma) = (1 + k)\,f - k\,(f * G_\sigma) \tag{3.23}$$

El parámetro k controla la intensidad del realce:

- $k = 0$: sin realce ($g = f$);
- $k = 1$: USM clásico — duplica la contribución de las altas frecuencias;
- $k > 1$: *High Boost Filtering* — amplificación más allá del doble, útil para imágenes muy borrosas.

Amplificación de ruido

El USM no distingue bordes de ruido — ambos son componentes de alta frecuencia. Para k elevado, el ruido presente en la imagen se amplifica junto con los bordes. Por ello, se recomienda aplicar una leve suavización antes del USM en imágenes ruidosas, o usar un σ pequeño en la gaussiana.

Para ilustrar las etapas del USM, la Figura 3.20 aplica el método a un *parche* 30×30 de la imagen del leopardo, utilizando $\sigma = 1$ y $k = 1$. Inicialmente, la imagen se suaviza mediante un filtro gaussiano. A continuación, la máscara de alta frecuencia se obtiene como la diferencia entre la imagen original y la suavizada. Finalmente, esta máscara se suma a la imagen original, reforzando bordes y detalles. La figura presenta las tres etapas del proceso y el resultado final del realce.

```
patch  = img_gray_crop[35:65, 45:75]
k = 1.0
sigma = 1.0

p = patch.astype(np.float32)

# Etapa 1: suavizado Gaussiano
ksize   = int(6 * sigma + 1) | 1  # operador binario garantiza tamaño impar
p_suave = mm.gaussian(patch, ksize, sigma).astype(np.float32)

# Etapa 2: máscara de alta frecuencia
mascara = p - p_suave

# Etapa 3: realce
p_usm = np.clip(p + k * mascara, 0, 255).astype(np.uint8)

# p_usm = mm.usm(patch, k)  # o, directo
```

```
print(f"Píxel central [1,1]: original={int(p[1,1])} suave={p_suave[1,1]:.1f}"
      f" máscara={mascara[1,1]:.1f} realzado={p_usm[1,1]}")

mm.show(
    [patch,
     p_suave.astype(np.uint8),
     np.clip(mascara + 128, 0, 255).astype(np.uint8),  # negativos visibles
     p_usm],
    titles=["Parche original",
            f"Suavizado ( ={sigma})",
            "Máscara (alta freq., +128)",
            f"Realzado USM (k={k})"],
    cols=4, figsize=(12, 4)
)
```

```
Píxel central [1,1]: original=16 suave=20.0 máscara=-4.0 realzado=12
```

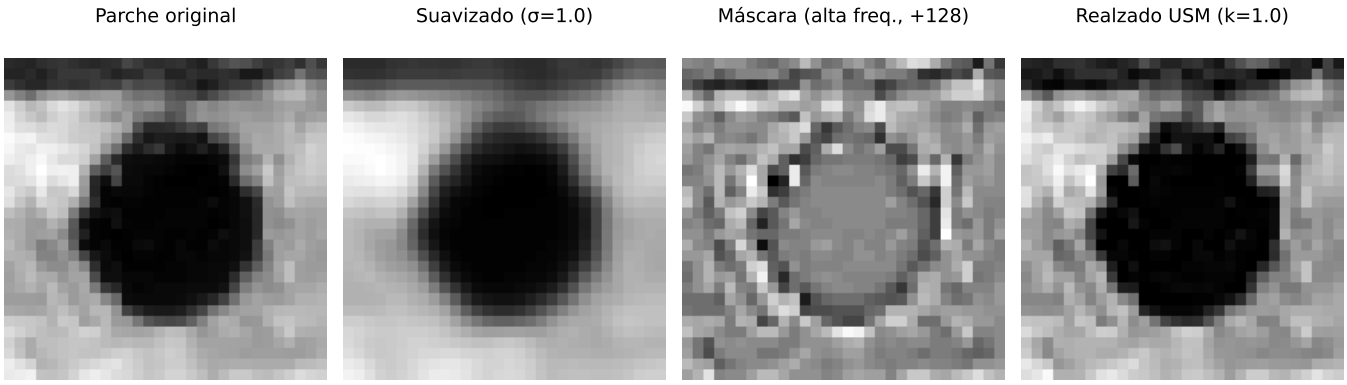


Figura 3.20: Realce de nitidez por *Unsharp Masking* en la imagen del leopardo: la imagen suavizada se resta de la original para generar la máscara de alta frecuencia, que luego se reintroduce para resaltar bordes y detalles.

La Figura 3.21 aplica el método USM a un recorte más grande de la imagen del leopardo utilizando $\sigma = 1$ y diferentes valores del factor de ganancia k. En todos los casos, la máscara de alta frecuencia se obtiene mediante la diferencia entre la imagen original y su versión suavizada por filtro gaussiano. El parámetro k controla la intensidad del realce: valores menores producen un aumento sutil de nitidez, mientras que valores mayores refuerzan progresivamente bordes y detalles. Se observa que, para valores elevados de k, surgen halos alrededor de los bordes y el ruido presente en la imagen comienza a amplificarse.

```
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
)
```

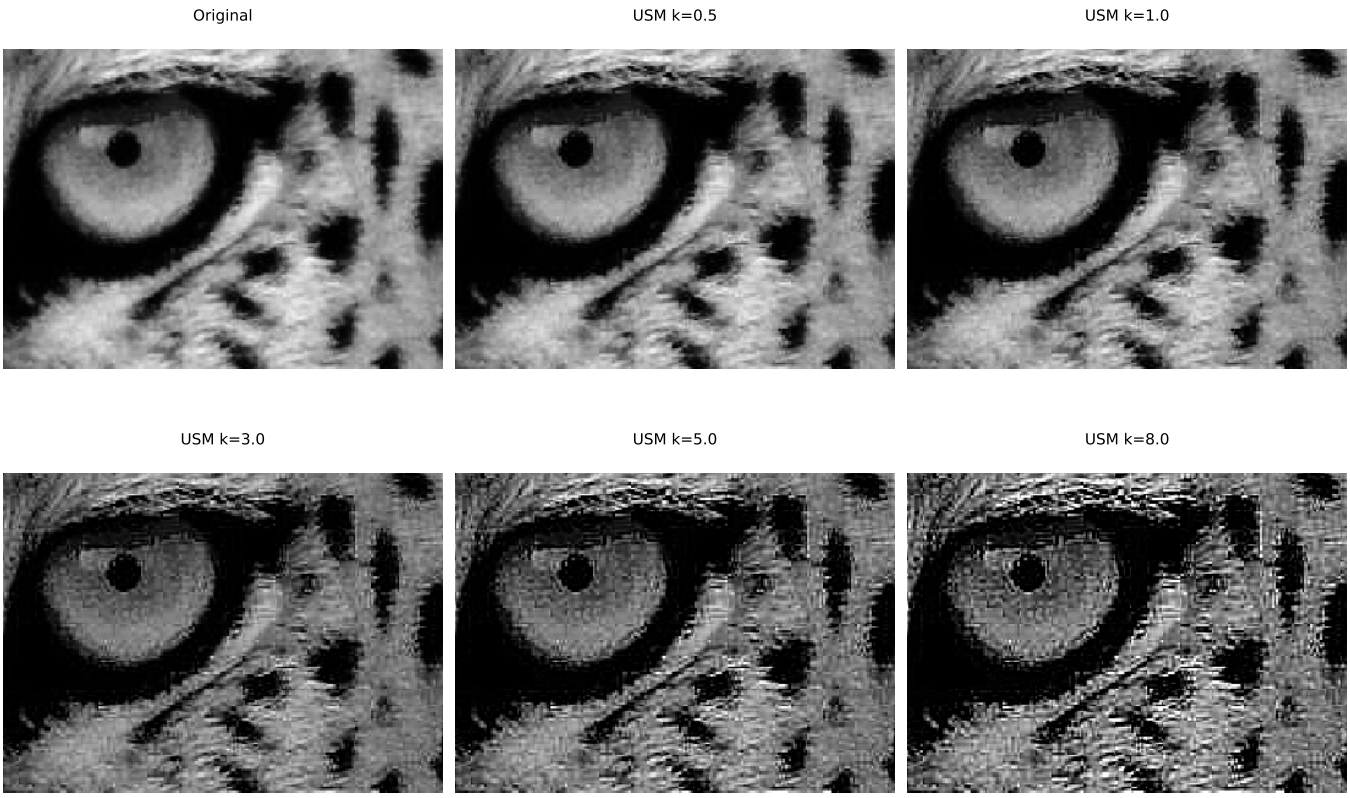


Figura 3.21: *Unsharp Masking* en la imagen del leopardo con =1 y k de 0.5 a 8.0. Para k>2 aparecen halos en los bordes y el ruido de fondo se nota.

3.6.5 Detector de Canny

Canny combina cuatro etapas en secuencia — suavizado Gaussiano, gradiente de Sobel, supresión de no-máximos e histéresis por doble umbral — para producir bordes **finos, binarios y conectados**. A diferencia de Sobel y Prewitt, el resultado no es un mapa de gradiente continuo, sino una máscara donde cada píxel es borde o no.

El parámetro central es el par de umbrales (T_{low}, T_{high}). Los píxeles con gradiente por encima de T_{high} son bordes seguros; por debajo de T_{low}, se descartan. Los píxeles ambiguos — entre los dos umbrales — se deciden mediante **histéresis**: se convierten en borde si están conectados a un borde seguro, y se descartan en caso contrario. Esto evita tanto la pérdida de tramos débiles de bordes reales como la inclusión de ruido aislado. Una heurística común es $T_{high} = 3 \times T_{low}$.

```
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
)
```

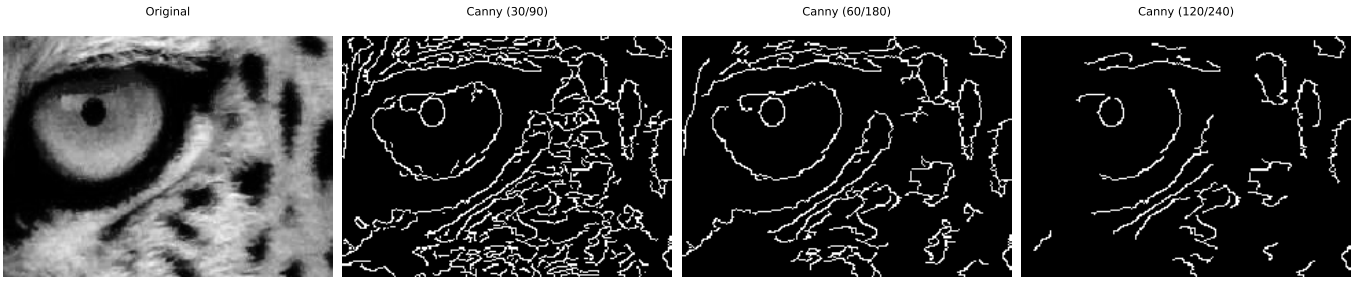


Figura 3.22: Detector de Canny con diferentes pares de umbral: umbrales bajos capturan más bordes (incluso ruido); umbrales altos retienen solo los bordes más fuertes.

Elección de los umbrales

Una heurística común es $T_{high} = 3 \times T_{low}$. Los valores típicos dependen del rango de gradiente de la imagen — `cv2.Canny` acepta valores absolutos en $[0, 255]$. Para imágenes con contraste variable, calcular los umbrales a partir de percentiles de la magnitud de Sobel es más robusto que usar valores fijos.

3.7 Filtros de Orden: Filtro de la Mediana

Los filtros de orden (*order-statistic filters*) reemplazan el píxel central por el valor de un **percentil** de la distribución de intensidades de la vecindad — a diferencia de los filtros lineales, que calculan combinaciones ponderadas. El más importante es el **filtro de la mediana**.

3.7.1 Ruido Impulsivo: Sal y Pimienta

El ruido **sal y pimienta** (*salt-and-pepper noise*) reemplaza píxeles aleatorios por valores extremos: 0 (pimienta, negro) o 255 (sal, blanco). Es común en la transmisión de imágenes con errores de bit y en cámaras con sensores defectuosos.

Para entender por qué fallan los filtros lineales, considere una vecindad 3×3 donde un único píxel ha sido corrompido a 255:

$$\text{vecindad} = \begin{bmatrix} 102 & 98 & 105 \\ 100 & \mathbf{255} & 97 \\ 103 & 99 & 101 \end{bmatrix}$$

Tabla 3.4: Media vs. mediana con un píxel corrompido. La mediana ignora el valor atípico; la media se desplaza ~40 niveles.

Método	Cálculo	Resultado
Media	(102+98+...+255+...+101)/9	140
Mediana	{97,98,99,100,**101**,102,103,105,255}	101

¿Por qué fallan los filtros de media con ruido impulsivo?

La media es sensible a los ***valores atípicos*** — un único píxel con valor 255 en una vecindad de valor 100 eleva la salida a 140, propagando el ruido por la imagen. La mediana, por ser un **estimador robusto**, selecciona el valor central de la distribución ordenada, descartando naturalmente los extremos sin ningún ajuste especial.

El siguiente ejemplo ilustra el comportamiento de la media y de la mediana en presencia de un píxel corrupto por ruido impulsivo. Se observa que la media está fuertemente influenciada por el valor extremo (255), produciendo una estimación distante de los valores predominantes de la vecindad. En cambio, la mediana permanece cercana al valor original de la región, evidenciando su mayor robustez frente a *valores atípicos* y justificando su uso en la eliminación de ruido de sal y pimienta.

```
# Ejemplo numérico: media vs. mediana con píxel corrupto
vizinhanca = np.array([102, 98, 105, 100, 255, 97, 103, 99, 101])
print(f"Vecindad: {sorted([int(x) for x in vizinhanca])}")
print(f"Media:      {vizinhanca.mean():.1f}  (desplazada por el valor atípico 255)")
print(f"Mediana:    {int(np.median(vizinhanca))}    (ignora el valor atípico)")
print(f"Valor original:  ~100")
```

```
Vecindad: [97, 98, 99, 100, 101, 102, 103, 105, 255]
Media:      117.8  (desplazada por el valor atípico 255)
Mediana:    101    (ignora el valor atípico)
Valor original:  ~100
```

La Figura 3.23 presenta el efecto del ruido sal y pimienta en diferentes densidades. El ruido fue generado reemplazando aleatoriamente una fracción de los píxeles por valores mínimos (0, pimienta) y máximos (255, sal). A medida que la densidad aumenta del 2% al 10%, crece la cantidad de píxeles corruptos, haciendo que la degradación visual sea más evidente y dificultando la percepción de los detalles de la imagen.

```
def add_salt_pepper(img, prob=0.05, seed=42):
    """Añade ruido sal y pimienta con probabilidad total prob."""
    noisy = img.copy()
    rnd   = np.random.default_rng(seed).random(img.shape)
    noisy[rnd < prob / 2]      = 0    # pimienta
    noisy[rnd > 1 - prob / 2] = 255  # sal
    n_corrompidos = int((rnd < prob/2).sum() + (rnd > 1-prob/2).sum())
    return noisy, n_corrompidos

probs = [0.02, 0.05, 0.10]
imgs_noise, titles_noise = [img_gray_crop], ["Original"]

for p in probs:
    noisy, n = add_salt_pepper(img_gray_crop, p)
    imgs_noise.append(noisy)
    titles_noise.append(f"Ruído {int(p*100)}%\n({n:,} pixels)")

mm.show(imgs_noise, titles=titles_noise, cols=4)
```

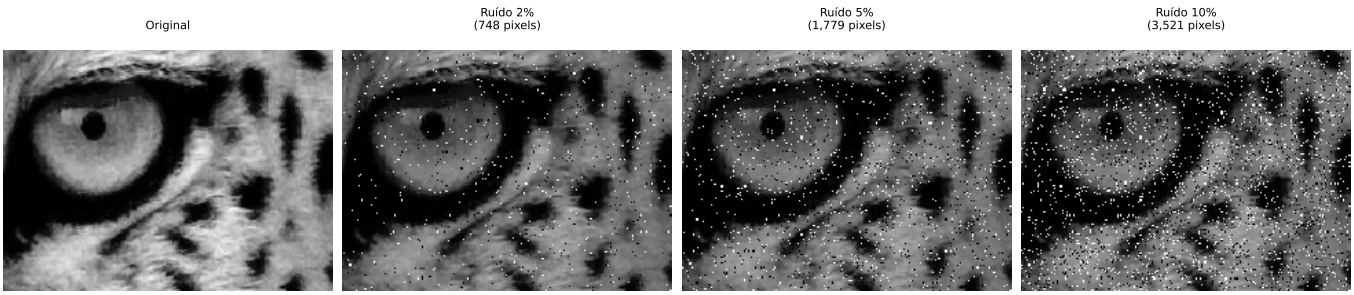


Figura 3.23: Ruido sal y pimienta con densidades crecientes (2%, 5%, 10%). El parámetro prob indica la fracción total de píxeles corrompidos, mitad sal (255) y mitad pimienta (0).

3.7.2 Filtro de la Mediana

El filtro de la mediana sustituye cada píxel por el **valor mediano** de los píxeles de su vecindad $n \times n$:

$$g(x, y) = \text{med}_{(s,t)\in\mathcal{V}_n}\{f(x + s, y + t)\} \tag{3.24}$$

El valor mediano es aquel que ocupa la posición central cuando los n^2 valores de la vecindad se ordenan. Para una ventana 3×3 ($n^2 = 9$ píxeles), la mediana es el 5º valor de la secuencia ordenada.

Para ilustrar, considere el mismo *patch* 5×5 con un píxel corrompido artificialmente en $[1, 1]$:

```
patch    = img_gray[250:255, 250:255].copy()
corrompido = patch.copy()
corrompido[1, 1] = 255   # inyecta píxel de sal en el centro

# Vecindad 3×3 centrada en [1,1]
viz_orig = sorted(patch[0:3, 0:3].ravel().tolist())
viz_corr = sorted(corrompido[0:3, 0:3].ravel().tolist())

print("Parche original:")
print(mm.drawImg(patch))
print("\nParche con píxel corrompido [1,1]=255:")
print(mm.drawImg(corrompido))
print(f"\nVecindad 3×3 original (ordenada):   {viz_orig}")
print(f"Mediana original:                      {viz_orig[4]}")
print(f"\nVecindad 3×3 corrompida (ordenada): {viz_corr}")
print(f"Mediana corrompida:                    {viz_corr[4]}  ← ignora el 255")
print(f"Media corrompida:                  {np.mean(viz_corr):.1f}  ← distorsionada por el
255")
```

```
Parche original:
 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


Parche con píxel corrompido [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


Vecindad 3×3 original (ordenada):   [91, 95, 102, 103, 106, 107, 107, 108, 108]
Mediana original:                       106

Vecindad 3×3 corrompida (ordenada): [91, 95, 102, 103, 106, 107, 108, 108, 255]
Mediana corrompida:                     106  ← ignora el 255
Media corrompida:                  119.4  ← distorsionada por el 255
```

El ejemplo confirma: incluso con el píxel corrompido a 255, la mediana devuelve el valor central correcto — el *outlier* ocupa la última posición en la ordenación y se descarta de forma natural.

Por basarse en la ordenación y no en la suma, la mediana posee tres propiedades fundamentales que la diferencian de los filtros lineales:

- **Robusta** ante el ruido impulsivo — los *outliers* van a los extremos de la secuencia ordenada y no afectan el valor central;
- **Preservadora de bordes** — las transiciones abruptas de intensidad se mantienen, pues la mediana selecciona un valor que ya existe en la vecindad, sin crear nuevos niveles intermedios;

- **No lineal** — no puede expresarse como convolución, por lo que `mm.conv` no se aplica; se utiliza `cv2.medianBlur`.

A Figura 3.24 compara diferentes técnicas de remoção de ruido sal y pimienta aplicadas a una imagen con un 10% de píxeles corruptos. Se evaluaron los filtros Gaussiano, Media, Mediana, Bilateral y Morfológico (próximo capítulo), lo que permite observar la relación entre la eliminación de ruido y la preservación de detalles. En general, los filtros de media y Gaussiano reducen el ruido, pero tienden a desenfocar los bordes, mientras que la mediana presenta un mejor rendimiento para ruido impulsivo. El filtro bilateral preserva mejor los bordes, y el filtro morfológico elimina gran parte de los píxeles corruptos sin degradar excesivamente la estructura de la imagen.

```
# 10% de ruido para evidenciar diferencias entre filtros
noisy_5, _ = add_salt_pepper(img_gray_crop, prob=0.1)

#   Filtros
f_gauss   = cv2.GaussianBlur(noisy_5, (5, 5), 1.0)
f_media   = mm.conv(noisy_5, np.ones((5, 5), dtype=np.float32) / 20.0)
f_median3 = cv2.medianBlur(noisy_5, 3)
f_median5 = cv2.medianBlur(noisy_5, 5)
f_bilat   = cv2.bilateralFilter(noisy_5, d=9, sigmaColor=75, sigmaSpace=75)

# filtros morfológicos en el próximo capítulo, pero ya adelantados aquí para comparación
# B         = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
# f_morf    = cv2.morphologyEx(           # presentado en el próximo capítulo
#                 cv2.morphologyEx(noisy_5, cv2.MORPH_OPEN,  B),
#                 cv2.MORPH_CLOSE, B)
B = mm.secross()  # elemento estructurante de caja 3×3
f_morf     = mm.asf(noisy_5, 'OC', B)  # equivalente vía morph

#   MSE
def mse(a, b):
    return float(np.mean((a.astype(np.float32) - b.astype(np.float32))**2))

print(f"{'Filtro':<22} {'MSE':>8}")
print("-" * 32)
pares = [("Gaussiano 5×5",    f_gauss),
         ("Média 5×5",        f_media),
         ("Mediana 3×3",      f_median3),
         ("Mediana 5×5",      f_median5),
         ("Bilateral",        f_bilat),
         ("Morf. open+close", f_morf)]
for nome, img in pares:
    print(f"{nome:<22} {mse(img_gray_crop, img):>8.2f}")

imgs   = [img_gray_crop, noisy_5,    f_gauss,        f_media,
          f_median3,     f_median5,  f_bilat,        f_morf]
titles = ["Original",    "Ruído 10%", "Gaussiano 5×5","Média 5×5",
          "Mediana 3×3", "Mediana 5×5", "Bilateral", "Morf. open+close"]

mm.show(
    imgs,
    titles=[f"Crop: {t}" for t in titles],
    cols=4, rows=2, figsize=(14, 8)
)
```

```
Filtro                      MSE
--------------------------------
Gaussiano 5×5            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
```

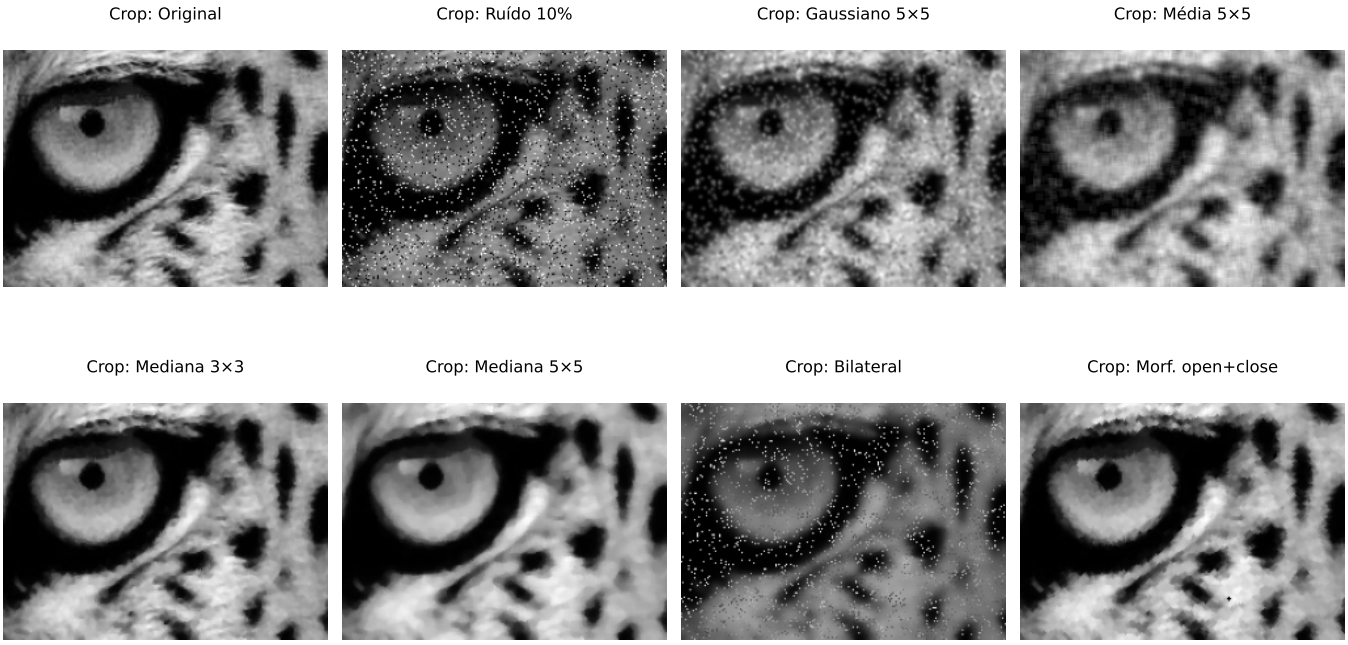


Figura 3.24: Comparación de filtros para eliminación de ruido sal y pimienta (10%): Gaussiano, Media, Mediana, Bilateral y Morfológico. Fila superior: imágenes completas; fila inferior: recorte de la región de interés.

3.8 Aplicación Práctica: Preprocesamiento para Segmentación

En la práctica, las técnicas de este capítulo rara vez se utilizan de forma aislada. Un ***pipeline* de preprocesamiento** típico combina varias etapas en secuencia, adaptándose al tipo de imagen y a la aplicación. La Figura 3.25 ilustra un *pipeline* completo:

1. **Ecualización de histograma (CLAHE):** normaliza el contraste independientemente de las condiciones de iluminación;
2. **Filtro Gaussiano:** suaviza el ruido de adquisición sin destruir bordes;
3. **Detección de bordes (Sobel/Canny):** extrae estructuras relevantes para la segmentación.

> **El orden importa**
>
> El orden de las operaciones afecta el resultado final. En general: **(1) normalización de intensidad → (2) reducción de ruido → (3) realce/segmentación**. Invertir el orden puede amplificar el ruido o perder bordes antes de detectarlos.

```python
# Etapa 1: CLAHE
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
img_clahe = clahe.apply(img_gray_crop)

# Etapa 2: Gaussiano
img_gauss = cv2.GaussianBlur(img_clahe, (5, 5), 0)

# Etapa 3: Canny
edges = cv2.Canny(img_gauss, 50, 150)
```

```
# Comparación: pipeline vs. Canny directo
edges_direct = cv2.Canny(img_gray_crop, 50, 150)

mm.show(
    [img_gray_crop, img_clahe, img_gauss, edges, edges_direct],
    titles=["Original", "1. CLAHE", "2. Gaussiano", "3. Canny (pipeline)", "Canny (directo)"],
    cols=5
)
```

Figura 3.25: *Pipeline* de preprocesamiento: CLAHE → Gaussiano → Canny. Cada etapa prepara la imagen para la siguiente, resultando en bordes limpios y bien definidos.

3.9 Resumen

En este capítulo se presentaron las principales técnicas de procesamiento en el dominio espacial, desde la manipulación directa de píxeles hasta el filtrado por vecindad:

- **Operaciones de punto:** aritméticas saturadas (`mm.addm`, `mm.subm`) y lógicas bit a bit (`mm.band`, `mm.bor`, `mm.bnot`) para recorte de ROI y combinación de imágenes; *alpha blending* (`mm.blend`) para fusión ponderada con peso $\alpha \in [0, 1]$.
- **Histograma:** función discreta de distribución de intensidades; visualizado con `mm.histImg` y calculado con `mm.hist`; base para el diagnóstico tonal y para las técnicas de ecualización y especificación.
- **Ecualización:** redistribución automática de las intensidades mediante la CDF (`mm.equalize`), con variante adaptativa CLAHE para el control local del contraste.
- **Especificación de histograma:** transferencia del perfil tonal de una imagen de referencia mediante mapeo inverso de la CDF — generalización de la ecualización para distribuciones arbitrarias.
- **Correlación y convolución:** mecanismo de ventana deslizante implementado en `mm.conv` (`cv2.filter2D`); diferenciados por la rotación de 180° del *kernel* — relevante solo para kernels asimétricos.
- **Filtros de suavizado:** media (*kernel* uniforme, desenfoca bordes proporcionalmente al tamaño) y Gaussiano (ponderación radial, separable, sin *ringing*, conserva mejor los bordes).
- **Filtros de realce:** Laplaciano (w_4/w_8, segunda derivada isotrópica), Sobel (gradiente direccional de primer orden, con magnitud $|\nabla f|$ y dirección θ) y *Unsharp Masking* (amplificación de las altas frecuencias con parámetro k).
- **Filtro de la mediana:** no lineal, robusto a valores atípicos, conserva bordes — superior a los filtros lineales para ruido sal y pimienta.
- **Pipeline práctico:** encadenamiento CLAHE → Gaussiano → Canny como estrategia de preprocesamiento; `mm.drawImgKernel` para visualización didáctica de la ventana deslizante.

El Capítulo 4 abordará la **morfología matemática** (erosión, dilatación, apertura y cierre), explorando en profundidad las funciones `mm.ero` y `mm.dil` de la biblioteca `morph.py`. A continuación, el Capítulo 5 presentará el **procesamiento en el dominio de la frecuencia**, con enfoque en la Transformada de Fourier y en técnicas de filtrado espectral.

3.10 🤖 Uso del Gemini Notebook como Tutor Complementario

En esta edición, incentivamos el uso del **Gemini Notebook** como herramienta complementaria de aprendizaje. Esta herramienta de IA utiliza exclusivamente los documentos proporcionados por el autor como base de conocimiento, garantizando respuestas coherentes con el contenido del libro — incluyendo las funciones de la biblioteca `morph.py` y los experimentos realizados en este capítulo.

Para cada capítulo, hemos preparado un proyecto específico en la plataforma con el PDF del capítulo, los *notebooks* y materiales auxiliares. Sugerimos explorar especialmente:

- **Guía de Estudio:** resumen estructurado de los conceptos, ideal para repasar antes de los exámenes;
- **Conversación:** resuelve dudas sobre ecualización, convolución, filtros y pipelines directamente con el tutor;
- **Preguntas frecuentes:** cuestiones típicas sobre la diferencia entre media y mediana, USM, Lapaciano vs. Sobel.

🎓 Estudia con el Tutor Inteligente

Para interactuar con el contenido de este capítulo, accede al siguiente *enlace*. El entorno contiene materiales didácticos en diferentes formatos, generados a partir del **PDF** del capítulo. En la plataforma, explora especialmente las opciones **Guía de Estudio** y **Conversación** para profundizar tu comprensión.
🚀 ACCEDER A GEMINI NOTEBOOK: CAPÍTULO 03

🌐 **Idioma y Lenguaje de Programación**

El proyecto de este capítulo en Gemini Notebook fue construido únicamente con el texto en **portugués** y los ejemplos de código en **Python**. Si estás estudiando con la edición en inglés o francés, o siguiendo la ruta en C++, las respuestas del tutor pueden no corresponder exactamente con la versión que estás leyendo.

⚠️ **Aviso sobre Contenido Generado por IA**

La IA es una poderosa aliada en los estudios, pero el contenido generado puede contener **errores o imprecisiones**. Consulta siempre **libros, artículos científicos y otras fuentes académicas confiables** para validar la información. Siempre que sea posible, ejecuta los ejemplos prácticos proporcionados en este capítulo para verificar los resultados.

3.11 Lista de Ejercicios

1. **(10%)** Explique la diferencia entre **convolución** y **correlación cruzada**. ¿Para qué tipos de *kernel* los resultados son idénticos? Dé un ejemplo de *kernel* asimétrico (como Sobel G_x) y muestre numéricamente que los resultados difieren aplicándolo al parche 5×5 del capítulo de las dos formas.

2. **(15%)** Considere una imagen 5×5 con intensidades concentradas entre los niveles 3 y 5 (bajo contraste, 3 bits). Aplique manualmente el algoritmo de ecualización de la Tabla 3.1, completando todas las columnas de la tabla (k, $h[k]$, $p[k]$, $\mathrm{cdf}[k]$, $\mathrm{lut}[k]$). Verifique el resultado con `mm.equalize`.

3. **(15%)** Usando `mm.conv`, aplique el filtro de media con kernels de tamaño 3×3, 9×9 y 21×21 a la imagen del mandril. Para cada versión, calcule el **PSNR** (*Peak Signal-to-Noise Ratio*) con respecto a la original:

$$\mathrm{PSNR} = 10\log_{10}\left(\frac{255^2}{\mathrm{MSE}}\right), \quad \mathrm{MSE} = \frac{1}{MN}\sum_{i,j}(f-g)^2$$

   Grafique el PSNR en función del tamaño del kernel y explique qué indica la caída progresiva sobre la relación entre suavizado y pérdida de información.

4. **(15%)** Usando `add_salt_pepper` con densidad del 5%, aplique y compare: (a) `mm.conv` con media 3×3, (b) `cv2.GaussianBlur` con $\sigma = 1$, (c) `cv2.medianBlur` con ventana 3×3 y (d) `cv2.medianBlur` con ventana 5×5. Muestre las imágenes con `mm.show` en una cuadrícula 2×4 (fila 1: imágenes, fila 2: histogramas mediante `mm.histImg`). Explique por qué la mediana supera a los filtros lineales usando el argumento de la Tabla 3.4.

5. **(15%)** Implemente `mm.conv0` usando solo operaciones NumPy vectorizadas — sin bucles en Python y sin `cv2.filter2D` — con el operador de *stride tricks* (`np.lib.stride_tricks.sliding_window_view`). Compare el resultado y el tiempo de ejecución con `mm.conv0` (bucles) y `mm.conv` (cv2) para kernels 3×3 y 15×15 en la imagen del mandril.

6. **(15%)** Aplique el *Unsharp Masking* con $\sigma = 1$ y $k \in \{0.5, 1.0, 2.0, 4.0\}$ usando la función `usm` del capítulo. Para cada valor de k: (a) calcule la diferencia absoluta $|g - f|$, (b) muestre las imágenes y las diferencias con `mm.show`, y (c) grafique el histograma de las diferencias con `mm.histImg`. Identifique a partir de cuál k los artefactos (halos y amplificación de ruido) se vuelven visualmente inaceptables.

7. **(15%)** Elija una imagen de rayos X o tomografía disponible públicamente (ej.: mediante `mm.read` de URL) y diseñe un pipeline de preprocesamiento con al menos 4 etapas secuenciales, justificando cada elección con base en los conceptos del capítulo. Muestre con `mm.show` en una cuadrícula: imagen original, cada etapa intermedia y el resultado final con sus histogramas (`mm.histImg`).

Referencias del Capítulo

La fundamentación teórica de este capítulo se basa en las siguientes obras:

- Gonzalez (2018) para los conceptos de operaciones de intensidad, histograma, convolución y filtrado espacial.
- Szeliski (2022) para la visión por computadora y aplicaciones prácticas de filtrado.
- Bradski (2008) para la implementación práctica con OpenCV y `morph.py`.

Executar Colab | Abrir GitHub

3.12 💻 Parte Práctica con Ejercicios de Programación

🎯 Objetivo de este Cuaderno

El cuaderno permite desarrollar, validar, organizar y probar soluciones de **Ejercicios de Programación (EPs)** en entornos interactivos, como Colab, con los mismos casos de prueba de Moodle, copiándolos allí solo al momento de registrar la nota oficial.

Download

Descargue `morph.py` y `testsuite.py` ejecutando la celda a continuación:

```
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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Ejecutando los Tests

Para evaluar los tests, ejecuta `TestSuite("EP03_01.extensión").run()` en una nueva celda, reemplazando la extensión por la del lenguaje utilizado (`.py`, `.java`, `.c`, `.cpp`, `.js` o `.r`). El sistema descarga los casos de prueba de GitHub, ejecuta el programa y calcula la nota automáticamente.

Para probar código Python directamente, sin guardar un archivo, usa `run_code(codigo)` pasando el código como *string* en una variable `codigo`:

```
codigo = """
from morph import mm
# ... tu código aquí ...
"""
TestSuite("EP03_01").run_code(codigo)
```

3.12.1 EP03_01 ➕ Adición Saturada de Constante

En sistemas de vigilancia por video, las cámaras en entornos con iluminación variable producen imágenes subexpuestas. El ajuste de brillo mediante **adición saturada de una constante** es la operación más simple para la corrección inmediata, aplicándose en tiempo real en los *chips* de cámaras embebidas y en *pipelines* de preprocesamiento de robots móviles.

Ver en Figura 3.26 una simulación de este EP.

3.12.1.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (filas) y C (columnas).
2. **Constante:** Leer el entero k (valor a sumar).
3. **Datos:** Leer los valores enteros de la matriz original fila por fila.
4. **Mapeo:** Para cada píxel p, calcular el nuevo valor mediante la ecuación:

$$p' = \text{clip}(p + k)$$

5. **Salida:** Mostrar la matriz resultante con dimensiones $L \times C$.

3.12.1.2 📌 Restricciones Computacionales

- **Saturación (*Clipping*):** Los valores deben confinarse al intervalo $[0, 255]$:

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

- **Tipo:** El resultado final debe ser entero (sin decimales).
- k **puede ser negativo:** los valores negativos oscurecen la imagen; los positivos la aclaran.

3.12.1.3 🧠 Fundamentación Teórica

Parámetro	Tipo	Impacto Visual
$k > 0$	Entero	Aclara la imagen; los píxeles cercanos a 255 saturan en blanco
$k < 0$	Entero	Oscurece la imagen; los píxeles cercanos a 0 saturan en negro
$k = 0$	Entero	Imagen sin cambios

3.12.1.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

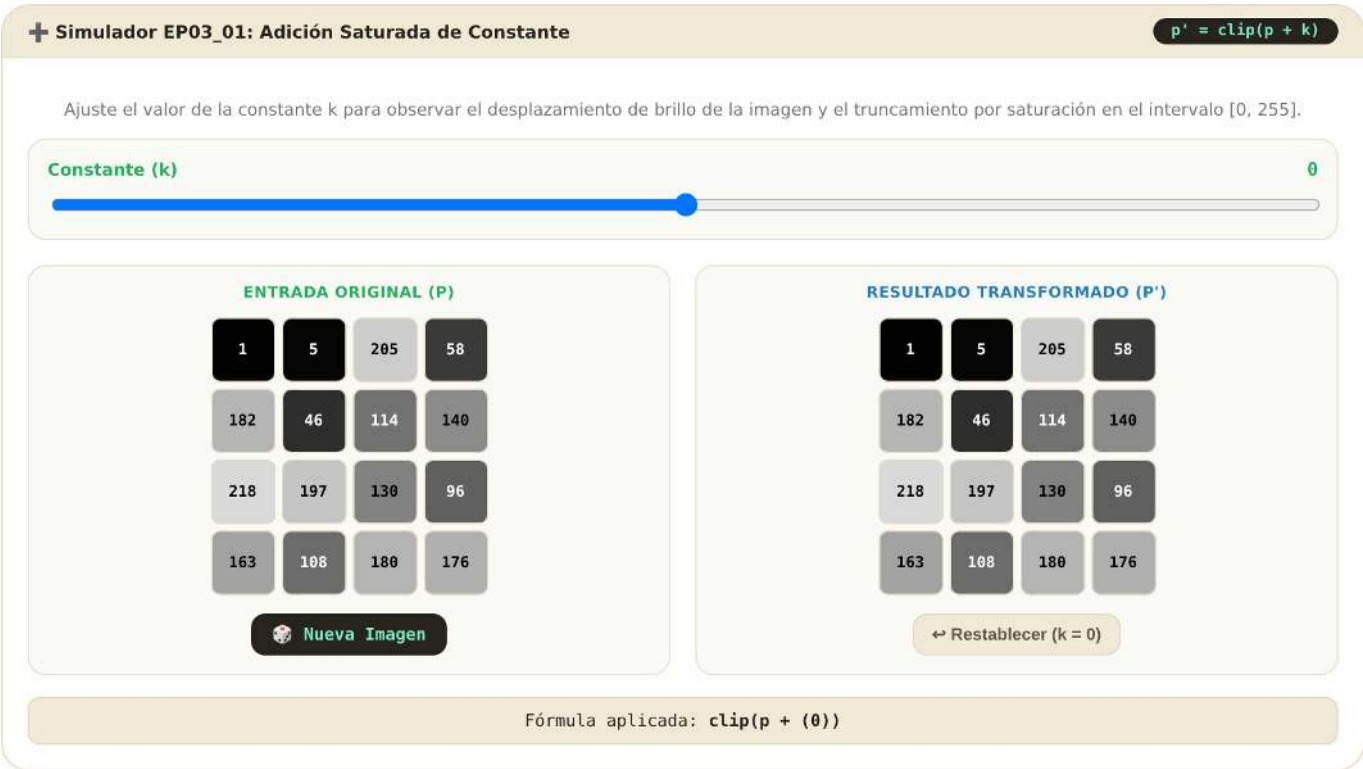
- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Entero k.
- Líneas siguientes: Elementos enteros de la matriz original.

Salida:

- Matriz transformada en L filas y C columnas, valores enteros separados por espacios.

3.12.1.5 📌 Ejemplos

Entrada	Salida	Observación
2 3 50 0 100 200 210 240 255	50 150 250 255 255 255	Saturación en 255 en los píxeles altos
1 4 -30 0 20 200 255	0 0 170 225	Saturación en 0 en los píxeles bajos



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap03/sim-ep0301-adicao.html

Figura 3.26: Simulador EP03_01: Adición Saturada de Constante (p' = clip(p + k))

```
%%writefile EP03_01.py
# Código Python
```

```
Overwriting EP03_01.py
```

```
TestSuite("EP03_01.py").run()
```

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

3.12.2 EP03_02 🔀 Alpha *Blending* de Dos Imágenes

En medicina nuclear, imágenes de diferentes modalidades (tomografía computarizada y resonancia magnética) se fusionan para ayudar en el diagnóstico. La **mezcla ponderada** (*alpha blending*) es la operación fundamental de este proceso, permitiendo al radiólogo controlar interactivamente el peso de cada modalidad en la imagen mostrada.

Ver en Figura 3.27 una simulación de este EP.

3.12.2.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (filas) y C (columnas).
2. **Parámetro:** Leer el valor real $\alpha \in [0, 1]$.
3. **Datos:** Leer los valores enteros de la matriz f_1 (imagen 1) y luego de la matriz f_2 (imagen 2).
4. **Mapeo:** Para cada posición (i, j), calcular:

$$g(i, j) = \text{clip}\left(\text{round}\left(\alpha \cdot f_1(i, j) + (1 - \alpha) \cdot f_2(i, j)\right)\right)$$

5. **Salida:** Mostrar la matriz resultante $L \times C$.

3.12.2.2 📌 Restricciones Computacionales

- **Redondeo:** Aplicar `round` antes de la conversión a entero.
- **Saturación:** Confinar al intervalo $[0, 255]$ con $\text{clip}(x) = \max(0, \min(255, x))$.
- **Operación en float:** Realizar la operación en punto flotante antes de redondear.

3.12.2.3 🧠 Fundamentación Teórica

Valor de α	Resultado
$\alpha = 1.0$	Solo f_1
$\alpha = 0.5$	Media aritmética de f_1 y f_2
$\alpha = 0.0$	Solo f_2

3.12.2.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

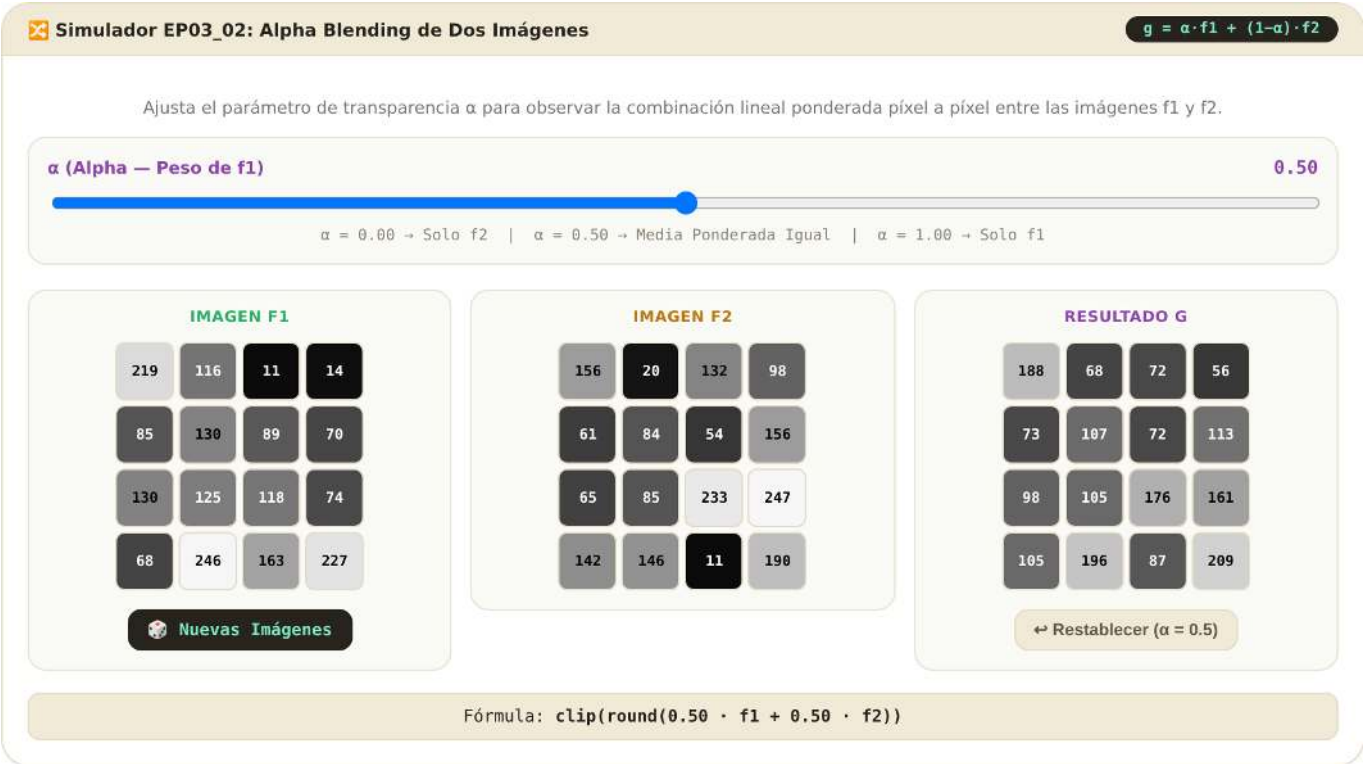
- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Real α.
- Líneas siguientes: Elementos de f_1 (L líneas con C valores cada una).
- Líneas siguientes: Elementos de f_2 (L líneas con C valores cada una).

Salida:

- Matriz resultante $L \times C$.

3.12.2.5 📌 Ejemplos

Entrada	Salida	Observación
1 3 0.5 0 100 200 100 200 50	50 150 125	Media entre las dos imágenes
1 3 1.0 10 20 30 90 80 70	10 20 30	Solo f_1 (alpha=1)



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap03/sim-ep0302-blending.html

Figura 3.27: Simulador EP03_02: Mezcla alfa de dos imágenes (g = · f1 + (1−) · f2)

```
%%writefile EP03_02.py
# Código Python
```

```
Overwriting EP03_02.py
```

```
TestSuite("EP03_02.py").run()
```

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

3.12.3 EP03_03 🎭 Inversión de Imagen (Negativo Fotográfico)

En radiología, las imágenes de rayos X se visualizan tradicionalmente en negativo: los huesos aparecen en negro sobre fondo blanco. La operación de **negativo fotográfico** se aplica rutinariamente en PACS (*Picture Archiving and Communication Systems*) para facilitar la detección de fracturas y densidades óseas.

Ver en Figura 3.28 una simulación de este EP.

3.12.3.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (filas) y C (columnas).
2. **Datos:** Leer los valores enteros de la matriz original.
3. **Mapeo:** Para cada píxel p, calcular el negativo:

$$p' = 255 - p$$

4. **Salida:** Mostrar la matriz resultante de $L \times C$.

3.12.3.2 📌 Restricciones Computacionales

- **Sin necesidad de *recorte*:** El resultado de $255 - p$ con $p \in [0, 255]$ siempre está $\in [0, 255]$.
- **Tipo entero:** La salida debe ser valores enteros.
- **Equivalencia lógica:** La operación es idéntica al `NOT` bit a bit (`mm.bnot`) en imágenes de 8 bits.

3.12.3.3 🧠 Fundamentación Teórica

Píxel Original p	Píxel Negativo p'	Observación
0 (negro)	255 (blanco)	Inversión total
128 (gris medio)	127 (gris medio)	Valor central
255 (blanco)	0 (negro)	Inversión total

3.12.3.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Líneas siguientes: Elementos enteros de la matriz original.

Salida:

- Matriz negativa en L filas y C columnas.

3.12.3.5 📌 Ejemplos

Entrada	Salida	Observación
140 128 200 255	255 127 55 0	Inversión de cada píxel
2 2 10 20 30 40	245 235225 215	Matriz 2x2 invertida

```
%%writefile EP03_03.py
# Código Python
```

```
Overwriting EP03_03.py
```

```
TestSuite("EP03_03.py").run()
```

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

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap03/sim-ep0303-inversao.html

Figura 3.28: Simulador EP03_03: Inversión de Imagen — Negativo Fotográfico (p' = 255 − p)

3.12.4 EP03_04 📊 Equalización de Histograma (L bits)

En imágenes de satélite de teleobservación, la variación de iluminación a lo largo del día produce imágenes de bajo contraste. La **equalización de histograma** se aplica automáticamente en satélites como el Landsat para redistribuir los tonos, revelando detalles de vegetación, relieve y zonas urbanas invisibles en la imagen original.

Ver en la Figura 3.29 una simulación de este EP.

3.12.4.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (líneas), C (columnas) y B (número de bits, con $L_{\max} = 2^B$).
2. **Datos:** Leer la matriz de píxeles f con valores en $[0, 2^B - 1]$.
3. **Histograma:** Calcular $h[k]$ = número de píxeles con intensidad k, para $k = 0 \dots 2^B - 1$.
4. **Probabilidad:** $p[k] = h[k]/(L \cdot C)$.
5. **CDF:** $\text{cdf}[k] = \sum_{j=0}^{k} p[j]$; función de distribución acumulada.
6. **LUT:** $\text{lut}[k] = \text{round}\,(\text{cdf}[k] \cdot (2^B - 1))$; *Look-Up Table* (tabla de consulta).
7. **Aplicación:** $g[i, j] = \text{lut}[f[i, j]]$.
8. **Salida:** Mostrar la matriz equalizada $L \times C$.

3.12.4.2 📌 Restricciones Computacionales

- **Redondeo:** Usar redondeo matemático (`round`) en la LUT.
- **Bits:** El número de niveles es 2^B (ej.: $B = 3 \Rightarrow 8$ niveles, $B = 8 \Rightarrow 256$ niveles).
- **CDF acumulada:** $\text{cdf}[k] = \sum_{j=0}^{k} p[j]$, con $\text{cdf}[2^B - 1] = 1.0$.

3.12.4.3 🧠 Fundamentación Teórica

Etapa	Operación	Fórmula
1	Histograma	$h[k] \leftarrow$ nº píxeles con intensidad k
2	Probabilidad	$p[k] = h[k]/(L \cdot C)$
3	CDF	$\text{cdf}[k] = \sum_{j=0}^{k} p[j]$

Etapa	Operación	Fórmula
4	LUT	$\text{lut}[Look-UpTable(tabladeconsulta)k] = \text{round}(\text{cdf}[k] \cdot (2^B - 1))$
5	Aplicación	$g[i,j] = \text{lut}[f[i,j]]$

3.12.4.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Entero B (número de bits).
- Líneas siguientes: Elementos enteros de la matriz.

Salida:

- Matriz equalizada en L líneas y C columnas.

3.12.4.5 📌 Ejemplos

Entrada	Salida	Observación
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	Ejemplo 3 bits del capítulo
1 4 3 0 0 7 7	0 0 7 7	Histograma bimodal extremo

```
%%writefile EP03_04.py
# Código Python
```

```
Overwriting EP03_04.py
```

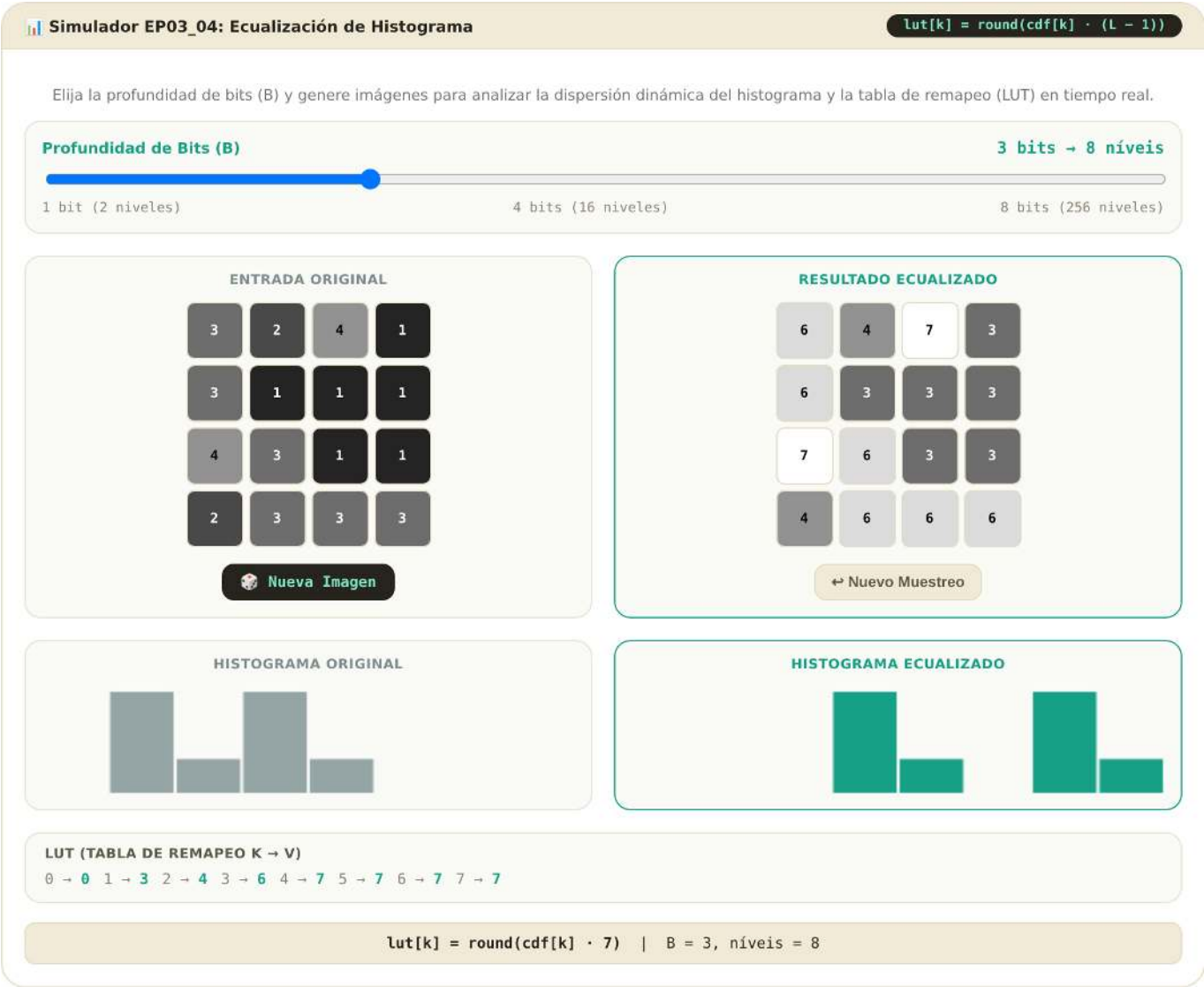
```
TestSuite("EP03_04.py").run()
```

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

3.12.5 EP03_05 🔲 Aplicación de Máscara AND Binaria

En sistemas de inspección industrial por visión por computadora, es necesario aislar regiones de interés (ROI) en imágenes de piezas para verificar defectos de fabricación. La operación **AND bit a bit con una máscara binaria** es el mecanismo fundamental para recortar exactamente el área de inspección, poniendo a cero todos los píxeles fuera de ella.

Ver en Figura 3.30 una simulación de este EP.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap03/sim-ep0304-equalizacao.html

Figura 3.29: Simulador EP03_04: Ecualización de Histograma (Niveles L = 2^B)

3.12.5.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (filas) y C (columnas).
2. **Datos:** Leer la matriz de píxeles f (valores $\in [0, 255]$).
3. **Máscara:** Leer la matriz binaria m (valores: solo 0 o 255).
4. **Mapeo:** Para cada píxel (i, j), aplicar el AND bit a bit:

$$g(i, j) = f(i, j) \text{ AND } m(i, j)$$

donde 255 = `11111111` y 0 = `00000000` en binario.

5. **Salida:** Mostrar la matriz resultante $L \times C$.

3.12.5.2 📌 Restricciones Computacionales

- **AND con 255:** p AND $255 = p$ (todos los bits preservados).
- **AND con 0:** p AND $0 = 0$ (todos los bits puestos a cero).
- **Máscara:** Los únicos valores posibles en la máscara son 0 y 255.
- **Implementación:** En Python, el AND bit a bit entre enteros usa el operador `&`.

3.12.5.3 🧠 Fundamentación Teórica

Píxel f	Máscara m	Resultado f AND m
cualquier v	255 (`11111111`)	v (preservado)
cualquier v	0 (`00000000`)	0 (puesto a cero)

3.12.5.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

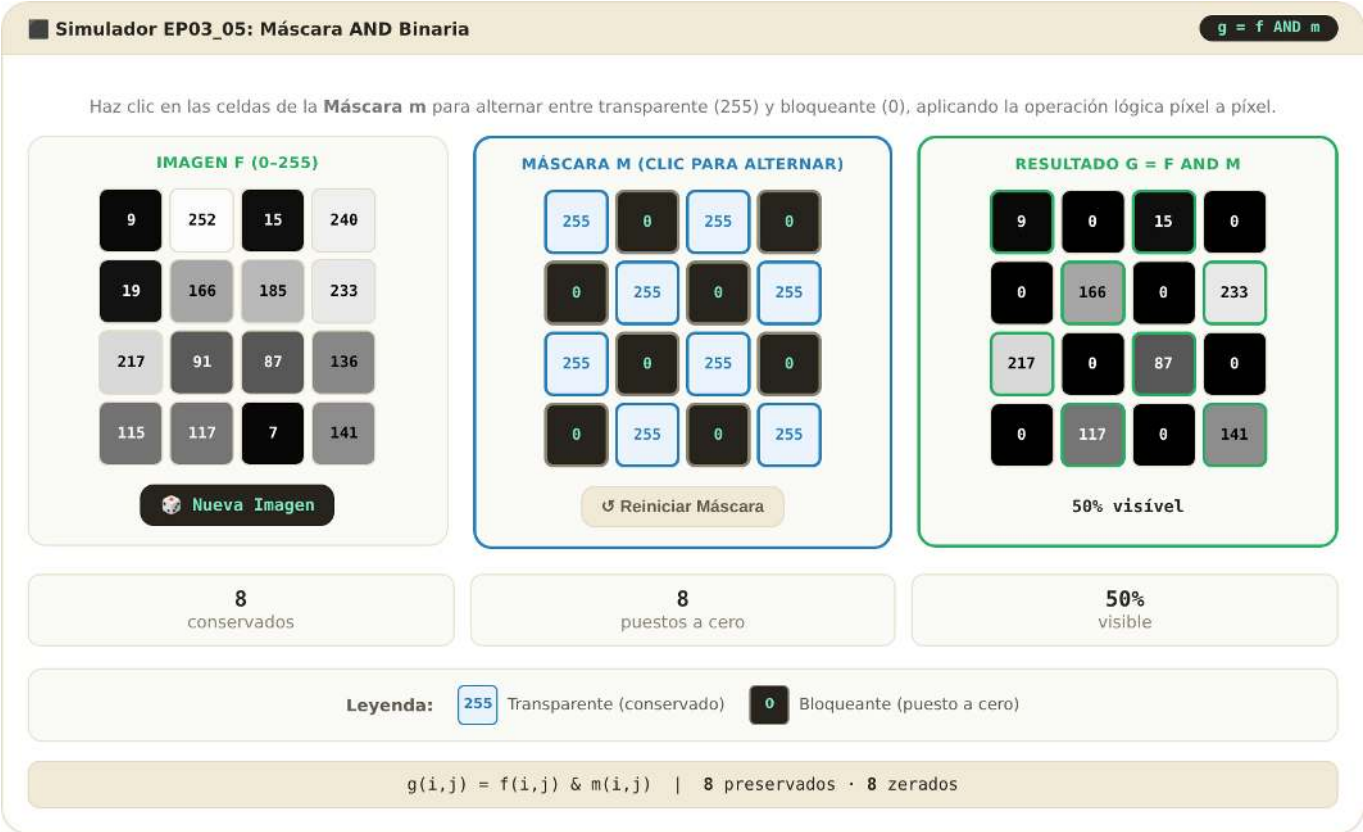
- Línea 1: Entero L.
- Línea 2: Entero C.
- Líneas siguientes: Elementos de f (L líneas).
- Líneas siguientes: Elementos de m (L líneas con valores 0 o 255).

Salida:

- Matriz resultante $L \times C$.

3.12.5.5 📌 Ejemplos

Entrada	Salida	Observación
2 3 100 150 200 50 80 120 255 255 0 0 255 255	100 150 0 0 80 120	La máscara selecciona la región
1 4 10 20 30 40 255 0 255 0	10 0 30 0	Alternado preservado/puesto a cero



Versión interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap03/sim-ep0305-mascara.html

Figura 3.30: Simulador EP03_05: Aplicación de Máscara AND Binaria

```
%%writefile EP03_05.py
# Código de Python
```

```
Overwriting EP03_05.py
```

```
TestSuite("EP03_05.py").run()
```

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

3.12.6 EP03_06 Filtro de Media con *Kernel* N×N

En cámaras de vehículos autónomos, las imágenes capturadas bajo lluvia o niebla presentan ruido gaussiano. El **filtro de media** se utiliza ampliamente para su reducción en tiempo real, implementándose directamente en el **ISP** (*Image Signal Processor*) de los sensores **CMOS** (*Complementary Metal-Oxide-Semiconductor*).

Los sensores CMOS son los sensores de imagen utilizados en la mayoría de las cámaras modernas (*smartphones*, *webcams*, cámaras automotrices, etc.). Convierten la luz en señales eléctricas, y el ISP procesa estas señales en tiempo real — aplicando operaciones como reducción de ruido, balance de blancos y otros ajustes de imagen.

Ver en la Figura 3.31 una simulación de este EP.

3.12.6.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (filas), C (columnas) y N (tamaño del *kernel*, siempre impar).
2. **Datos:** Leer la matriz de píxeles f.
3. **Filtro de Media:** Para cada píxel (i, j) **interno** (sin bordes), calcular:

$$g(i,j) = \text{round}\left(\frac{1}{N^2}\sum_{s=-(r)}^{r}\sum_{t=-(r)}^{r} f(i+s,\, j+t)\right), \quad r = \lfloor N/2 \rfloor$$

4. **Tratamiento de Bordes:** Los píxeles en el borde (donde la ventana $N \times N$ sobrepasa los límites) deben **copiarse directamente** del original sin modificación.
5. **Salida:** Mostrar la matriz resultante $L \times C$.

3.12.6.2 📌 Restricciones Computacionales

- **Radio:** $r = \lfloor N/2 \rfloor$ (mitad del *kernel*, entero).
- **Píxeles internos:** (i,j) con $r \leq i < L - r$ y $r \leq j < C - r$.
- **Redondeo:** Usar redondeo matemático antes de convertir a entero.
- **Sin *clipping*:** El promedio de valores $\in [0, 255]$ permanece en $[0, 255]$.

3.12.6.3 🧠 Fundamentación Teórica

Tamaño N	Coeficiente	Píxeles en la ventana	Efecto
3	$1/9 \approx 0.111$	9	Suave
5	$1/25 = 0.04$	25	Medio
7	$1/49 \approx 0.020$	49	Fuerte

3.12.6.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

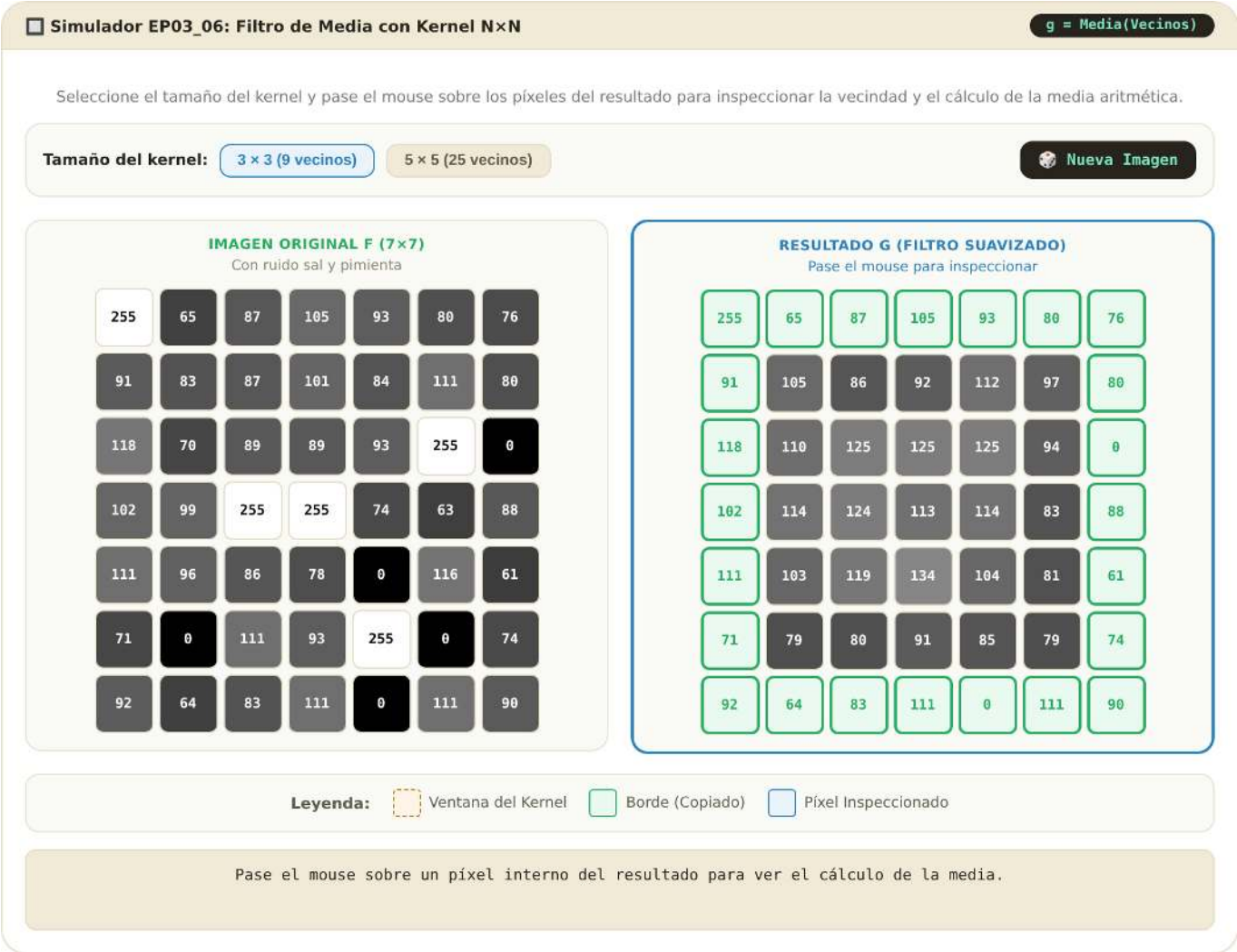
- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Entero N (impar, $N \geq 3$).
- Líneas siguientes: Elementos de la matriz original.

Salida:

- Matriz filtrada $L \times C$.

3.12.6.5 📌 Ejemplos

Entrada	Salida	Observación
3 3 3 10 20 30 40 50 60 70 80 90	10 20 30 40 50 60 70 80 90	Solo borde (3×3 = borde total)
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	Píxel aislado: todos los 9 píxeles internos cuya ventana 3×3 incluye el valor 100 reciben round(100/9)=11



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap03/sim-ep0306-media.html

Figura 3.31: Simulador EP03_06: Filtro de Media con Kernel N×N

```
%%writefile EP03_06.py
# Código Python
```

```
Overwriting EP03_06.py
```

```
TestSuite("EP03_06.py").run()
```

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

3.12.7 EP03_07 🔍 Operador Laplaciano (w4) para Realce de Bordas

En tomografías de alta resolución, la nitidez de los bordes entre tejidos es crítica para el diagnóstico. El **operador Laplaciano** se utiliza ampliamente en *pipelines* de preprocesamiento de imágenes médicas para resaltar automáticamente los contornos anatómicos antes de la segmentación, evitando la intervención manual del radiólogo.

Ver en Figura 3.32 una simulación de este EP.

3.12.7.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (filas) y C (columnas).
2. **Datos:** Leer la matriz de píxeles f.
3. **Laplaciano (w4):** Para cada píxel **interno** (i, j) con $1 \leq i < L - 1$, $1 \leq j < C - 1$, calcular:

$$\nabla^2 f(i, j) = f(i-1, j) + f(i+1, j) + f(i, j-1) + f(i, j+1) - 4 \cdot f(i, j)$$

4. **Realce:** Calcular la imagen realzada:

$$g(i, j) = \text{clip}(f(i, j) - \nabla^2 f(i, j))$$

5. **Borde:** Los píxeles en el borde se copian directamente: $g(i, j) = f(i, j)$.
6. **Salida:** Mostrar la matriz realzada $L \times C$.

3.12.7.2 📌 Restricciones Computacionales

- ***Kernel* w4:** $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ — solo vecinos-4.
- **Saturación:** $\text{clip}(x) = \max(0, \min(255, x))$ aplicado al resultado del realce.
- **Sin redondeo:** El Laplaciano utiliza solo sumas/restas de enteros.

3.12.7.3 🧠 Fundamentación Teórica

Región	$\nabla^2 f$	Efecto del Realce
Uniforme	≈ 0	Sin alteración
Borde creciente	< 0	Píxel aclarado
Borde decreciente	> 0	Píxel oscurecido

3.12.7.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.

- Líneas siguientes: Elementos de la matriz original.

Salida:

- Matriz realzada $L \times C$.

3.12.7.5 📌 Ejemplos

Entrada	Salida	Observación
3 3 0 0 0 0 100 0 0 0 0	0 0 0 0 255 0 0 0 0	Pico aislado: 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	Región uniforme: Laplaciano=0, sin alteración

```
%%writefile EP03_07.py
# Código Python
```

```
Overwriting EP03_07.py
```

```
TestSuite("EP03_07.py").run()
```

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

3.12.8 EP03_08 🧭 Gradiente de Sobel: Gx y Gy

En robots exploradores de Marte (como el Perseverance), la detección de obstáculos se realiza en tiempo real mediante cámaras estereoscópicas. El **operador de Sobel** calcula el gradiente direccional de la escena y se utiliza en el algoritmo de detección de bordes para identificar rocas, fisuras y desniveles del terreno que puedan comprometer la navegación.

Ver en Figura 3.33 una simulación de este EP.

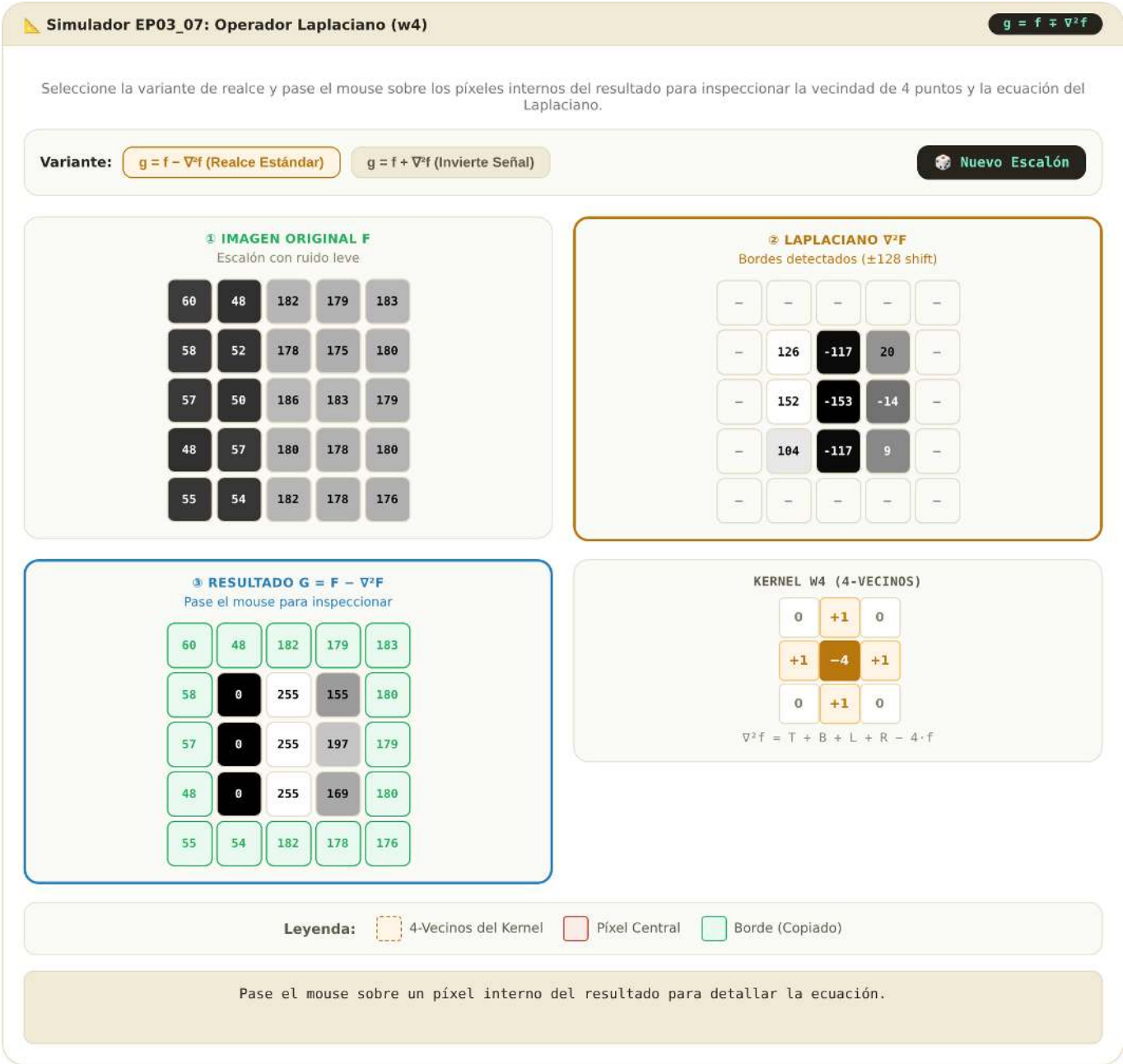
3.12.8.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (filas) y C (columnas).
2. **Datos:** Leer la matriz f.
3. **Gx y Gy:** Para cada píxel **interno** (i, j) con $1 \leq i < L-1$, $1 \leq j < C-1$:

$$G_x(i,j) = [f(i-1,j+1) + 2f(i,j+1) + f(i+1,j+1)] - [f(i-1,j-1) + 2f(i,j-1) + f(i+1,j-1)]$$

$$G_y(i,j) = [f(i+1,j-1) + 2f(i+1,j) + f(i+1,j+1)] - [f(i-1,j-1) + 2f(i-1,j) + f(i-1,j+1)]$$

4. **Magnitud:** $|\nabla f(i,j)| = \text{clip}(\text{round}(\sqrt{G_x^2 + G_y^2}))$.
5. **Borde:** Los píxeles de borde reciben magnitud 0.
6. **Salida:** Mostrar la magnitud $L \times C$.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap03/sim-ep0307-laplaciano.html

Figura 3.32: Simulador EP03_07: Operador Laplaciano (w4) para Realce de Bordas

3.12.8.2 📌 Restricciones Computacionales

- **Redondeo:** Aplicar `round` antes de convertir a entero.
- **Saturación:** $\text{clip}(x) = \max(0, \min(255, x))$.
- **Raíz cuadrada:** Usar $\sqrt{G_x^2 + G_y^2}$ (no la aproximación $|G_x| + |G_y|$).

3.12.8.3 🧠 Fundamentación Teórica

Operador	Detecta	Coeficientes diagonales
G_x	Bordes verticales	± 1
G_y	Bordes horizontales	± 1
$\|\nabla f\|$	Todos los bordes	Combinado

3.12.8.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Líneas siguientes: Elementos de la matriz.

Salida:

- Magnitud del gradiente, matriz $L \times C$.

3.12.8.5 📌 Ejemplos

Entrada	Salida	Observación
3 3 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0	Imagen nula: gradiente cero
3 3 0 0 255 0 0 255 0 0 255	0 0 0 0 255 0 0 0 0	Borde vertical central: Gx alto

```
%%writefile EP03_08.py
# Código Python
```

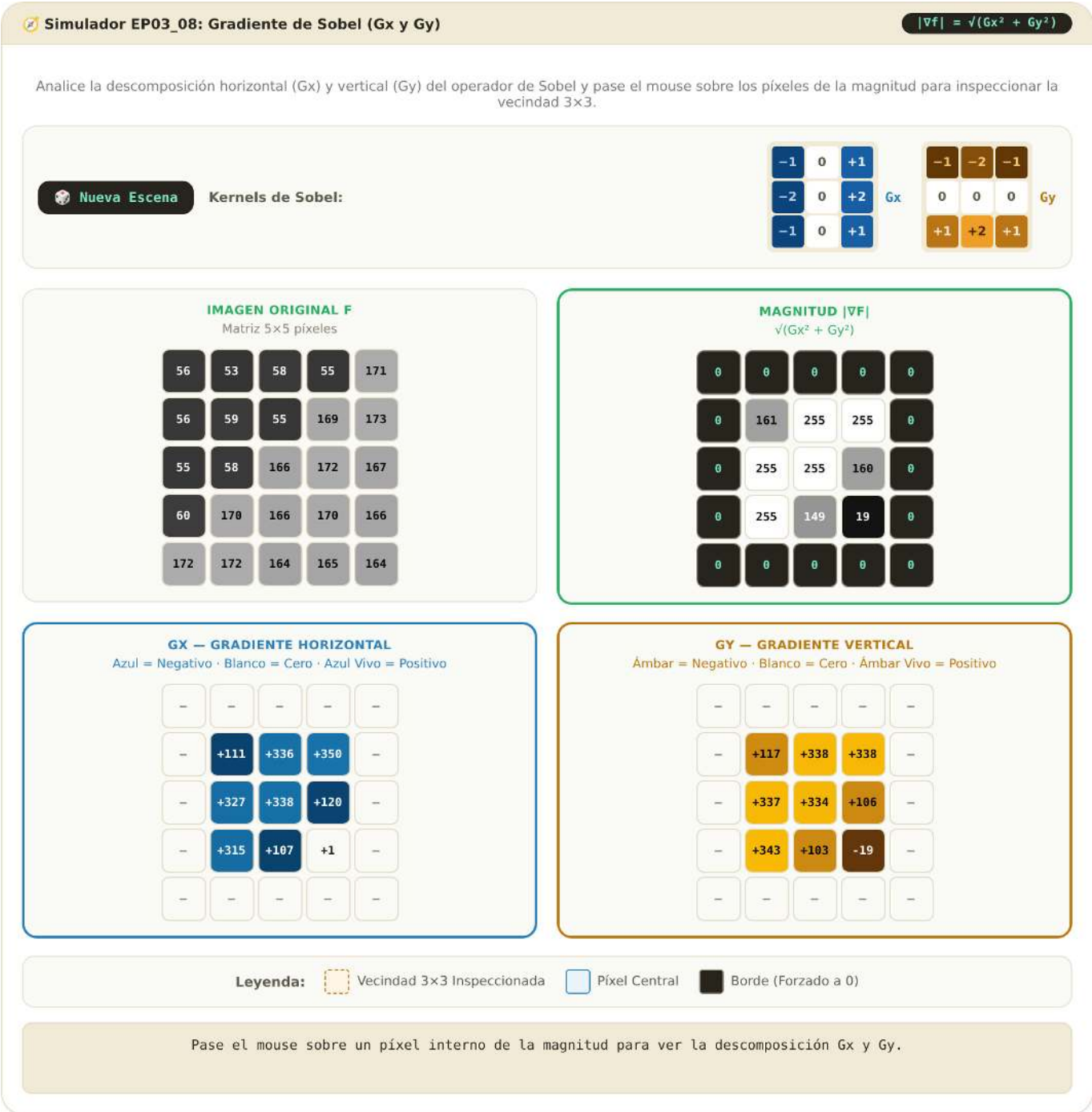
```
Overwriting EP03_08.py
```

```
TestSuite("EP03_08.py").run()
```

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

3.12.9 EP03_09 📡 Filtro de la Mediana 3×3

Las imágenes de radar de apertura sintética (SAR) utilizadas en monitoreo ambiental y militar sufren de un tipo específico de ruido llamado *speckle*, que posee características similares al ruido sal y pimienta. El **filtro**



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap03/sim-ep0308-sobel.html

Figura 3.33: Simulador EP03_08: Gradiente de Sobel (Gx y Gy)

de la mediana es el método estándar para eliminar este ruido porque preserva los bordes de las estructuras mientras elimina los puntos espurios.

Ver en Figura 3.34 una simulación de este EP.

3.12.9.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (filas) y C (columnas).
2. **Datos:** Leer la matriz de píxeles f.
3. **Filtro de la Mediana 3×3:** Para cada píxel **interno** (i, j) con $1 \leq i < L-1$, $1 \leq j < C-1$:
   - Recolectar los 9 píxeles de la vecindad 3×3: $\{f(i+s, j+t) : s, t \in \{-1, 0, 1\}\}$.
   - Ordenar los 9 valores en orden creciente.
   - Asignar $g(i, j)$ al valor central (posición índice 4, considerando índice 0).

$$g(i, j) = \text{mediana}\{f(i+s, j+t) : s, t \in \{-1, 0, 1\}\}$$

4. **Borde:** Copiar directamente: $g(i, j) = f(i, j)$.
5. **Salida:** Mostrar la matriz filtrada $L \times C$.

3.12.9.2 📌 Restricciones Computacionales

- **Ventana:** Siempre $3 \times 3 = 9$ elementos.
- **Mediana:** El elemento central de la secuencia ordenada (índice 4 de 0 a 8).
- **Sin recorte:** La mediana de valores en $[0, 255]$ permanece en $[0, 255]$.
- **No lineal:** El filtro de mediana no puede expresarse como convolución lineal.

3.12.9.3 🧠 Fundamentación Teórica

Ruido	Filtro de Media	Filtro de Mediana
Sal y pimienta (0 o 255)	Dispersa el ruido	Elimina sin distorsionar bordes
Gaussiano	Reduce eficazmente	Reduce parcialmente

3.12.9.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Líneas siguientes: Elementos de la matriz.

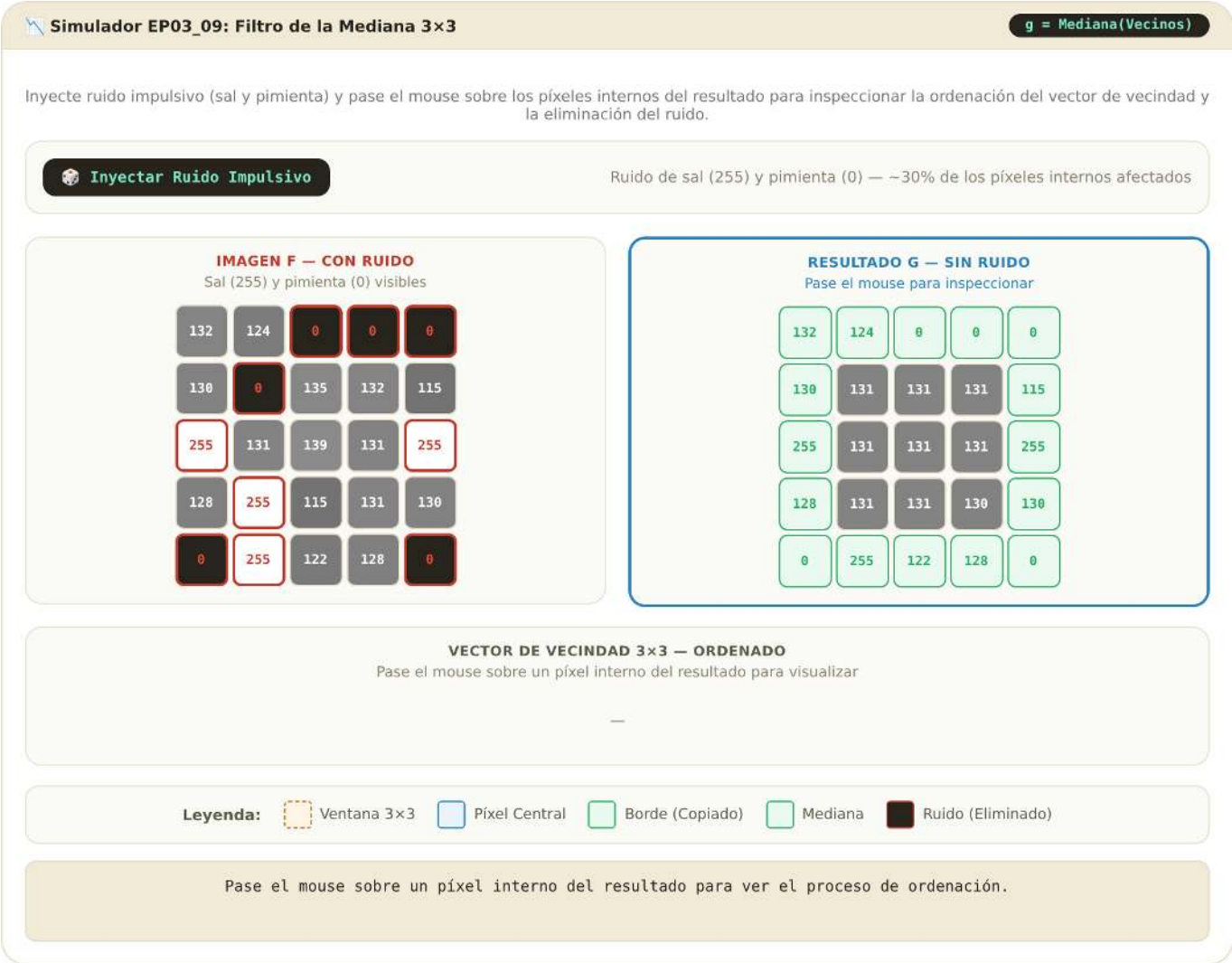
Salida:

- Matriz filtrada $L \times C$.

3.12.9.5 📌 Ejemplos

Entrada	Salida	Observación
3 3 100 100 100 100 0 100 100 100 100	100 100 100 100 100 100 100 100 100	Punto negro eliminado: mediana de $8{\times}100{+}1{\times}0 = 100$

Entrada	Salida	Observación
3 3 50 50 50 50 255 50 50 50 50	50 50 50 50 50 50 50 50 50	Punto blanco (sal) eliminado



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap03/sim-ep0309-mediana.html

Figura 3.34: Simulador EP03_09: Filtro de la Mediana 3×3

```
%%writefile EP03_09.py
# Código Python
```

```
Overwriting EP03_09.py
```

```
TestSuite("EP03_09.py").run()
```

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

3.12.10 EP03_10 ✨ *Unsharp Masking* (USM)

En los sistemas de digitalización de documentos históricos y obras de arte, la nitidez de las imágenes es fundamental para la lectura de textos manuscritos y detalles ornamentales. El ***Unsharp Masking*** **(USM)** es el algoritmo de realce de nitidez estándar utilizado en *escáneres* profesionales y software como Adobe Photoshop, controlado por el parámetro k que determina la intensidad del realce.

Ver en la Figura 3.35 una simulación de este EP.

3.12.10.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (líneas) y C (columnas).
2. **Parámetro:** Leer el valor real k (intensidad del realce, $k \geq 0$).
3. **Datos:** Leer la matriz de píxeles f.
4. **Suavizado:** Calcular $\bar{f}$ con el filtro de promedio 3×3 (solo píxeles internos; bordes mantenidos):

$$\bar{f}(i,j) = \frac{1}{9} \sum_{s=-1}^{1} \sum_{t=-1}^{1} f(i+s, j+t)$$

5. **Máscara de alta frecuencia:** $m(i,j) = f(i,j) - \bar{f}(i,j)$.
6. **Realce USM:** Para cada píxel interno:

$$g(i,j) = \text{clip}\left(\text{round}\left(f(i,j) + k \cdot m(i,j)\right)\right)$$

7. **Borde:** $g(i,j) = f(i,j)$ (copia directa).
8. **Salida:** Mostrar la matriz realzada $L \times C$.

3.12.10.2 📌 Restricciones Computacionales

- **Redondeo:** Aplicar `round` antes del clip.
- **Saturación:** $\text{clip}(x) = \max(0, \min(255, x))$.
- **Operaciones en float:** Calcular $\bar{f}$ y m en punto flotante antes de redondear el resultado final.
- $k = 0$**:** Sin realce — la salida es idéntica a la entrada (excepto en los bordes).

3.12.10.3 🧠 Fundamentación Teórica

Etapa	Operación	Descripción
1	$\bar{f} = f * \frac{1}{9}\mathbf{1}_{3\times3}$	Suavizado (bajas frecuencias)
2	$m = f - \bar{f}$	Máscara (altas frecuencias)
3	$g = \text{clip}(\text{round}(f + k \cdot m))$	Realce ponderado

3.12.10.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Real k.
- Líneas siguientes: Elementos de la matriz original.

Salida:

- Matriz realzada $L \times C$.

3.12.10.5 📌 Ejemplos

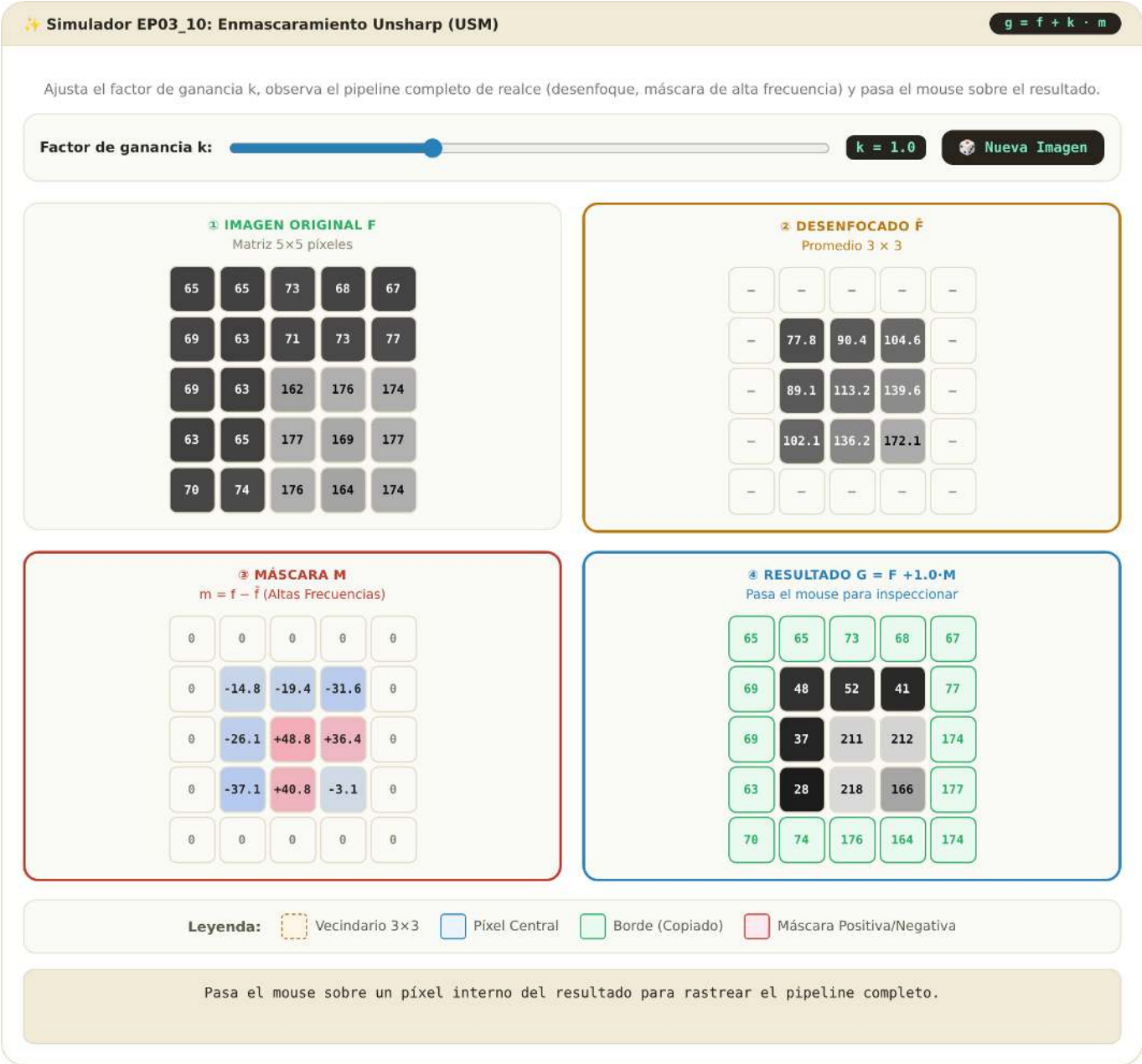
Entrada	Salida	Observación
3 3 0.0 100 100 100 100 100 100 100 100 100	100 100 100 100 100 100 100 100 100	k=0: sin realce
3 3 1.0 50 50 50 50 200 50 50 50 50	50 50 50 50 255 50 50 50 50	k=1: píxel central realzado y saturado

```
%%writefile EP03_10.py
# Código Python
```

```
Overwriting EP03_10.py
```

```
TestSuite("EP03_10.py").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap03/sim-ep0310-unsharp.html

Figura 3.35: Simulador EP03_10: *Máscara de enfoque* (USM)

Capítulo 4

Morfología Matemática y Segmentación de Imágenes

Executar Colab Abrir GitHub

Este capítulo presenta dos temas fundamentales del Procesamiento Digital de Imágenes (PDI): la **morfología matemática** y la **segmentación de imágenes**. La morfología matemática proporciona un marco teórico basado en la teoría de conjuntos para analizar, refinar y cuantificar la forma de objetos en imágenes binarias y en tonos de gris, mediante operadores fundamentales como la **erosión** y la **dilatación**. La segmentación, por su parte, tiene como objetivo particionar la imagen en regiones de interés, separando objetos del fondo y produciendo representaciones adecuadas para el análisis y la interpretación.

El capítulo comienza con la **umbralización**, una de las técnicas más importantes de segmentación, introduciendo el método automático de **Otsu** y revisitando el análisis de histogramas mediante la varianza interclases, presentada en el Capítulo 1. A continuación, se estudian los principales operadores de la morfología matemática, incluyendo erosión, dilatación, apertura, cierre y reconstrucción morfológica, que permiten refinar máscaras binarias y preservar estructuras relevantes de los objetos. Finalmente, se presentan técnicas de segmentación basada en regiones, como el **etiquetado de componentes conexos**, la **transformada de distancia** y el algoritmo ***watershed*** basado en marcadores, culminando en la extracción de descriptores geométricos y en la generación de *bounding boxes* compatibles con sistemas modernos de detección de objetos.

4.1 Objetivos

Al final de este capítulo, usted será capaz de:

- **Aplicar umbralización:** Comprender el criterio automático de Otsu por maximización de la varianza interclases (σ_B^2) y seleccionar estrategias adecuadas de preprocesamiento para facilitar la segmentación;
- **Dominar la morfología binaria:** Comprender y aplicar erosión ($A \ominus B$) y dilatación ($A \oplus B$) como operadores fundamentales, derivando apertura ($A \circ B$), cierre ($A \bullet B$) y operaciones basadas en reconstrucción morfológica, como `mm.clohole` y `mm.edgeoff`;
- **Aplicar morfología en tonos de gris:** Utilizar gradiente morfológico y filtros *top-hat* para realce y análisis de estructuras locales;
- **Etiquetar componentes conexos:** Identificar y separar regiones conectadas en imágenes binarias mediante algoritmos de etiquetado;
- **Aplicar transformada de distancia:** Interpretar y calcular distancias al fondo utilizando enfoques morfológicos y métricas geométricas;
- **Segmentar por regiones:** Construir *pipelines* de segmentación basados en marcadores utilizando Transformada de Distancia y el algoritmo ***watershed***;
- **Extraer descriptores geométricos:** Calcular propiedades como área, perímetro, centroide, circularidad y *bounding boxes* mediante `mm.label0` y extracción de contornos;
- **Relacionar PDI y visión computacional:** Comprender cómo los descriptores extraídos por segmentación pueden convertirse a formatos utilizados por detectores modernos, como 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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0
```

4.2 Umbralización

La **umbralización** (*thresholding*) es una de las formas más simples y eficientes de segmentación de imágenes. Su objetivo es clasificar cada píxel en dos clases de intensidad, normalmente asociadas a *objeto* y *fondo*:

$$g(x,y) = \begin{cases} 255, & \text{si } f(x,y) > T \\ 0, & \text{en caso contrario} \end{cases} \tag{4.1}$$

donde $f(x,y)$ representa la intensidad del píxel en la imagen original y $g(x,y)$ la imagen binaria resultante.

La elección del umbral T es importante para la calidad de la segmentación. El método de **Otsu** determina automáticamente el umbral óptimo al maximizar la **varianza entre clases** σ_B^2 definida por:

$$\sigma_B^2(T) = w_0(T)\, w_1(T)\, \left[\mu_0(T) - \mu_1(T)\right]^2 \tag{4.2}$$

donde:

- $w_0(T)$ y $w_1(T)$ son las probabilidades acumuladas de las clases fondo y objeto;
- $\mu_0(T)$ y $\mu_1(T)$ son las medias de intensidad de esas clases;
- $\sigma_B^2(T)$ representa la varianza entre clases para un umbral dado T.

El método funciona mejor cuando el histograma presenta dos grupos de intensidades relativamente separados. Para ello, el algoritmo evalúa todos los umbrales posibles de la imagen — típicamente en el intervalo $[0, 255]$ para imágenes de 8 bits — y selecciona el valor que maximiza la varianza entre clases, denotada por σ_B^2:

$$T^* = \arg \max_{T \in [0,255]} \sigma_B^2(T)$$

ℹ Otsu asume histogramas bimodales

El método de Otsu produce mejores resultados cuando el histograma presenta dos picos bien definidos (*bimodalidad*), correspondientes al fondo y al objeto. Cuanto mayor sea la separación entre estos picos y más pronunciado sea el máximo de σ_B^2, más confiable tiende a ser el umbral obtenido.
En imágenes con iluminación no uniforme o múltiples regiones de intensidad, las técnicas de **umbralización adaptativa** — en las cuales el umbral se calcula localmente — suelen producir segmentaciones más robustas.
El subíndice B en σ_B^2 significa ***between classes*** (*entre clases*). Así, σ_B^2 representa la **varianza entre las clases** (*between-class variance*).

4.2.1 Imagen de Monedas

La imagen utilizada para practicar la segmentación es una fotografía de una colección de monedas de diferentes países y épocas (Figura 4.1). Crédito: GAZI.MD.AHAD (CC BY-SA 4.0). Presenta objetos circulares con bordes bien definidos, siendo ideal para demostrar umbralización, operadores morfológicos, transformada de distancia, *watershed* y descriptores de forma.

```
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"Dimensiones [y,x,c]: {img_coins_color.shape}")
mm.show(img_coins_color, scale=30)
```

```
Dimensiones [y,x,c]: (2560, 1920, 3)
```

Figura 4.1: Imagen con monedas de varios tipos. Crédito: GAZI.MD.AHAD (CC BY-SA 4.0).

4.2.2 Preprocesamiento para el Otsu

La calidad del método de Otsu depende directamente de cuán **bimodal** es el histograma de la imagen de entrada. La imagen original de las monedas presenta iluminación no uniforme y monedas oscuras (cobre oxidado) con intensidades cercanas a las del fondo oscuro, lo que hace que el histograma sea poco bimodal.

Para minimizar estas limitaciones, se evaluarán dos técnicas clásicas de preprocesamiento, presentadas en el capítulo anterior, aplicadas antes de la etapa de umbralización. La Tabla 4.1 resume las características de

cada enfoque. Estas técnicas buscan aumentar la separación entre objeto y fondo, haciendo que el histograma se aproxime más a una distribución bimodal.

Tabla 4.1: Técnicas de preprocesamiento evaluadas para mejorar la separación entre monedas y fondo antes de la aplicación del método de Otsu.

Técnica	Qué hace	Cuándo usar
CLAHE	Ecualización de histograma adaptativa local	Bajo contraste global o regional
Gaussiano	Suavizado por convolución con gaussiana	Ruido de alta frecuencia (textura del fondo)

La función `cv2.createCLAHE(clipLimit, tileGridSize)` divide la imagen en bloques y aplica ecualización de histograma en cada uno de ellos, limitando la amplificación del ruido mediante el parámetro `clipLimit`. Por su parte, el filtro Gaussiano (`cv2.GaussianBlur`) suaviza texturas finas que podrían crear picos falsos en el histograma.

Para comparar objetivamente qué preprocesamiento produce la mejor entrada para el método de Otsu, se presentarán, para cada versión de la imagen, la propia imagen, el histograma con el umbral óptimo T^* destacado, la curva $\sigma_B^2(T)$ y el resultado de la binarización.

Criterio de comparación

La versión que presente el mayor valor de σ_B^2 en su pico proporciona la mejor separación entre las clases de fondo y objeto y, en consecuencia, la mejor entrada para el método de Otsu (Figura 4.2).

```
import cv2, io
import matplotlib.pyplot as plt

def otsu_criterio(img):
    h=mm.hist(img); p=h/h.sum();                  # histograma y probabilidades
    sigma2=np.zeros(len(p))
    for T in range(1,len(p)):                     # recorre umbrales
        w0,w1=p[:T].sum(),p[T:].sum()             # probabilidades de las clases
        if w0*w1==0: continue                     # evita división por cero
        mu0=(np.arange(T)*p[:T]).sum()/w0         # media fondo
        mu1=(np.arange(T,len(p))*p[T:]).sum()/w1  # media objeto
        sigma2[T]=w0*w1*(mu0-mu1)**2              #  ²B(T)
    return sigma2,np.argmax(sigma2)               # curva y T óptimo

def fig2img(fig):
    b=io.BytesIO(); fig.savefig(b,format='png',dpi=100)   # figura → buffer
    plt.close(fig); b.seek(0)
    return np.array(plt.imread(b))                        # buffer → array

def plot_curve(y,T,title,ylabel,color):
    fig,ax=plt.subplots(figsize=(4,3))
    ax.plot(y,color=color) if ylabel=="  ²B" else ax.bar(range(len(y)),y,color=color,width=1)
    ax.axvline(T,color='red',lw=2,label=f"T*={T}")       # umbral óptimo
    ax.set(xlabel="T" if ylabel=="  ²B" else "Intensidade",ylabel=ylabel)
    ax.legend(fontsize=8); plt.tight_layout()
    return fig2img(fig)

#   Preprocesamientos
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)]

#   Tabla comparativa
print(f"{'Versão':<18}{'T*':>6}{' ²B pico':>14}")
print("-"*40)

imgs,titles=[],[]
for nome,img in imgs0:
    sigma2,T=otsu_criterio(img)                          # calcula  ²B(T)
    print(f"{nome:<18}{T:>6}{sigma2[T]:>14.4e}")
    imgs += [
        img,                                             # imagen
        plot_curve(mm.hist(img),T,f"Hist T*={T}","Freq.","steelblue"),
        plot_curve(sigma2,T," ²B(T)"," ²B","darkorange"),
        mm.threshold(img)                               # Otsu final
    ]
    titles += [nome,f"Hist T*={T}"," ²B(T)",f"Otsu T*={T}"]

#   Exhibición final
mm.show(imgs,titles=titles,cols=4,figsize=(12,12),dpi=200)
```

```
Versão                 T*       ²B pico
----------------------------------------
Original              105    1.9437e+03
CLAHE                 122    2.4695e+03
CLAHE+Gauss           123    2.4283e+03
```

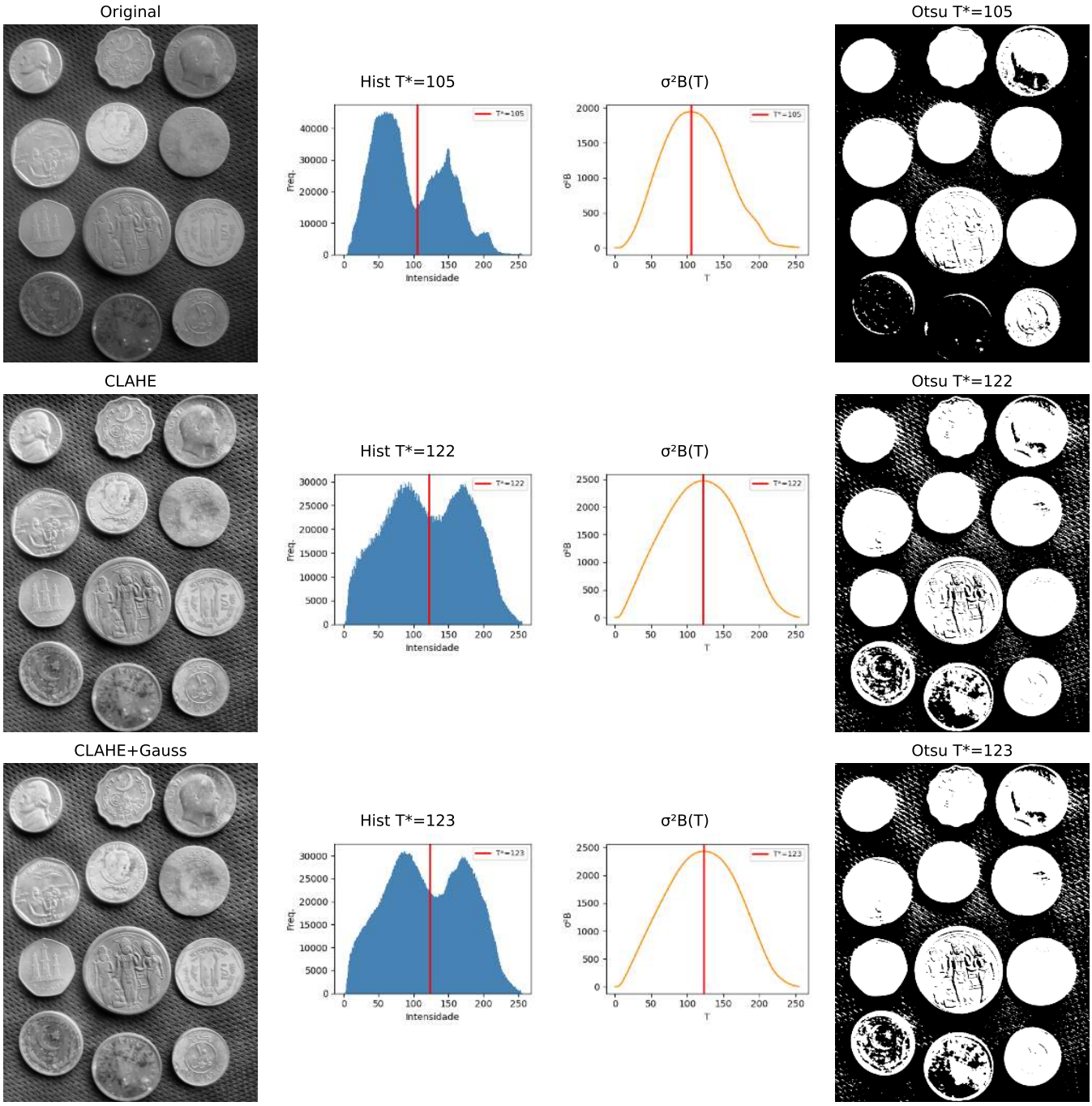


Figura 4.2: Comparación de los preprocesamientos: imagen | histograma+T* | ²B(T) | Otsu. La mejor separación bimodal indica el umbral más confiable.

4.2.3 Resultado: CLAHE como Mejor Preprocesamiento

El análisis de la Figura 4.2 indica que el **CLAHE** obtuvo el mayor valor de la varianza entre clases ($\sigma_B^2 \approx 2{,}47 \times 10^3$), con umbral óptimo $T^* = 122$. Aunque la combinación **CLAHE+Gaussiano** haya producido un resultado muy similar ($\sigma_B^2 \approx 2{,}43 \times 10^3$, $T^* = 123$), el criterio cuantitativo del método de Otsu favorece ligeramente el uso del CLAHE de forma aislada.

En términos visuales, las imágenes binarizadas obtenidas con CLAHE y CLAHE+Gaussiano son prácticamente equivalentes. La diferencia entre ambos enfoques se hace más evidente en el análisis de los histogramas y de los valores de $\sigma_B^2(T)$ que en la inspección directa de las segmentaciones resultantes. Así, la elección del CLAHE se basa principalmente en la maximización de la separación estadística entre las clases de fondo y objeto.

Interpretación de los resultados

Observe que los preprocesamientos con CLAHE y CLAHE+Gaussiano producen histogramas y umbrales óptimos muy próximos ($T^* = 122$ y $T^* = 123$). En consecuencia, las imágenes binarizadas resultantes también son bastante similares. En este caso, la decisión no se basa en diferencias visuales marcadas, sino en el criterio objetivo del método de Otsu: el mayor valor de σ_B^2 indica la mejor separación entre las clases.

4.3 Morfología Matemática

La **morfología matemática** es una teoría basada en conjuntos utilizada para analizar la forma y la estructura de objetos en imágenes. A diferencia de los filtros lineales presentados en el Capítulo 3, los operadores morfológicos son **no lineales**, pues se basan en operaciones de mínimo, máximo e inclusión espacial, en lugar de combinaciones lineales de intensidades. Estos operadores actúan sobre la vecindad de cada píxel mediante un **elemento estructurante** $\mathbb{B}$, responsable de definir la forma y el tamaño de la región analizada.

En imágenes binarias y en tonos de gris con elementos planos, el elemento estructurante trasladado a la posición x se define espacialmente como:

$$\mathbb{B}_x = \{x + b \mid b \in \mathbb{B}\}$$

En las regiones de borde de la imagen, parte del conjunto $\mathbb{B}_x$ puede extrapolar el dominio físico de la escena ($\mathbb{E}$). Para garantizar la consistencia matemática de los operadores primitivos en esas fronteras, se asume teóricamente que el espacio exterior al dominio de la imagen se rellena con el **elemento neutro** de la operación correspondiente (infinito positivo para la erosión e infinito negativo para la dilatación), impidiendo que el entorno externo corrompa las estructuras internas del objeto.

Cuando el elemento estructurante asocia pesos a sus elementos — es decir, $b : \mathbb{B} \to \mathbb{Z}$ — se denomina **función estructurante** o **elemento estructurante no plano**.

Desarrollada por Matheron y Serra en la década de 1960 para imágenes binarias y posteriormente extendida a tonos de gris, la morfología matemática fundamenta operadores como el gradiente morfológico, el *top-hat*, el *watershed* y la transformada de distancia, todos derivados de dos primitivos: la **erosión** y la **dilatación** [Matheron (1975); Serra (1982)].

4.3.1 Erosión y Dilatación

Los dos operadores primitivos se definen de manera unificada para imágenes en tonos de gris ($f : \mathbb{E} \to \mathbb{Z}$) y, por restricción al dominio $\{0, 1\}$, también para imágenes binarias.

4.3.1.1 Erosión

La **Erosión** de una imagen f por una función estructurante $b : \mathbb{B} \to \mathbb{Z}$ se define formalmente por:

$$\varepsilon_b(f)(x) = (f \ominus b)(x) = \min_{z \in \mathbb{B}} \{\, f(x + z) - b(z) \,\}, \quad \forall\, x \in \mathbb{E} \tag{4.3}$$

En la práctica, la erosión sustituye la intensidad del píxel x por el valor mínimo resultante de la diferencia entre la imagen y el elemento estructurante en la vecindad definida por el dominio $\mathbb{B}$. Los valores positivos en los pesos de $b(z)$ fuerzan el resultado local hacia abajo, "excavando" el relieve de la imagen más profundamente e intensificando la erosión.

En el caso **plano** (donde los pesos son nulos dentro del dominio, es decir, $b \equiv 0$), la expresión se simplifica al mínimo local puro:

$$\varepsilon_B(f)(x) = \min\{\, f(y) : y \in \mathbb{B}_x \,\}$$

En imágenes binarias, esta operación equivale a exigir que el conjunto $\mathbb{B}$, trasladado a la coordenada x, esté **completamente contenido** en el objeto A:

$$A \ominus \mathbb{B} = \{\, z \in \mathbb{E} \mid \mathbb{B}_z \subseteq A \,\}$$

Efecto Visual: *Encoge* objetos y estructuras claras, eliminando protuberancias, picos brillantes o ruidos que sean geométricamente menores que el dominio $\mathbb{B}$.

4.3.1.2 Implementación de la erosión

La versión didáctica `mm.ero0` implementa el caso particular de erosión con **elemento estructurante plano**. Para cada píxel (y, x), la función recorre los vecinos espaciales permitidos por B y almacena el menor valor encontrado en la imagen de entrada f, reproduciendo directamente la operación de mínimo local descrita en la Ecuación 4.3 para $b \equiv 0$.

Observe que los valores de los vecinos siempre se leen de forma estática de la imagen original f; la matriz de salida g se utiliza exclusivamente para registrar el mínimo acumulado de la vecindad actual. De esta forma, el resultado final es invariante respecto al orden de barrido de los píxeles (ya sea por filas o columnas).

La función auxiliar `_viz` calcula las coordenadas de los vecinos válidos dentro de los límites físicos de la imagen. En los bordes, la inicialización del acumulador en `255` emula con exactitud el relleno por elemento neutro exigido por la teoría. Por su parte, la función de interfaz `mm.ero` recurre a la implementación nativa y optimizada de OpenCV (`mm.ero`) cuando el elemento estructurante es plano, cambiando a la rutina general `mm.ero1` en caso de que el elemento posea pesos topográficos.

El ejemplo computacional siguiente ilustra la aplicación de un elemento estructurante en cruz (`mm.secross()`) destacado en la Figura 4.3, comparando la ejecución de la variante didáctica en bucle (`mm.ero0`) con el motor computacional de OpenCV (`mm.ero`).

```
B_cruz = mm.secross()
mm.drawImgPlt(B_cruz, scale=40)
```

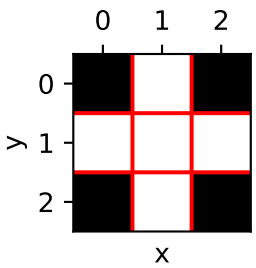


Figura 4.3: Elemento estructurante en forma de cruz (B_{cruz}) utilizado para conectividad-4.

```
def _viz(f, B, y, x):
    """Genera (vy, vx, b_val) para cada vecino válido de (y,x)."""
    H, W = f.shape
    Bh, Bw = B.shape
    oh, ow = -Bh/2 + 0.5, -Bw/2 + 0.5  # offsets fijos
    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]
```

```python
def ero(f, Bc=np.zeros((3,3),dtype='uint8')):
    """Erosión (OpenCV o con pesos)."""
    try:    return cv2.erode(f, Bc)
    except: return mm.ero1(f, Bc)

def ero0(f, Bc=np.ones((3,3),dtype='uint8')):
    """Erosión clásica sin pesos."""
    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

# Script de Prueba y Validación
B = mm.secross()
print("Elemento estructurante B:")
print(mm.drawImage(B))

f = mm.randomImage(5,5)
print("Imagen original f:")
print(mm.drawImage(f))

print("Erosión Plana Didáctica (ero0):")
print(mm.drawImage(ero0(f, B)))

print("Erosión Optimizada OpenCV (ero):")
print(mm.drawImage(ero(f, B)))
```

```
Elemento estructurante B:
0 1 0
1 1 1
0 1 0

Imagen original f:
9 8 5 2 5
0 8 0 8 0
2 9 9 0 2
3 8 0 3 2
9 1 3 2 3

Erosión Plana Didáctica (ero0):
0 5 0 2 0
0 0 0 0 0
0 2 0 0 0
2 0 0 0 2
1 1 0 2 2

Erosión Optimizada OpenCV (ero):
0 5 0 2 0
0 0 0 0 0
0 2 0 0 0
2 0 0 0 2
1 1 0 2 2
```

La función `_viz` utiliza `yield` para generar cada vecino bajo demanda, sin almacenar todos los resultados en memoria en una lista. En el experimento siguiente, una ventana de 3000×3000 produce 9 millones de vecinos. La implementación basada en lista consumió más de 1 GB de RAM y tomó aproximadamente 46 s, mientras que la versión con `yield` consumió memoria despreciable y se ejecutó en 38 s. En procesamiento de imágenes, los generadores son especialmente útiles para recorrer grandes vecindarios de manera eficiente.

```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 Dilatación

La **Dilatación** de una imagen f mediante una función estructurante $b : \mathbb{B} \to \mathbb{Z}$ se define formalmente como:

$$\delta_b(f)(x) = (f \oplus b)(x) = \max_{z \in \mathbb{B}} \{ f(x - z) + b(z) \}, \quad \forall\, x \in \mathbb{E} \tag{4.4}$$

En la práctica, la dilatación reemplaza la intensidad del píxel x por el mayor valor resultante de la suma entre la imagen y el elemento estructurante en la vecindad definida. El argumento de inversión espacial $(x - z)$ indica que la dilatación evalúa implícitamente el elemento transpuesto (reflejado) $\hat{b}$, propiedad fundamental para asegurar la dualidad matemática respecto a la erosión.

En el caso **plano** (donde los pesos son nulos dentro del dominio, es decir, $b \equiv 0$), la expresión se reduce al máximo local puro:

$$\delta_B(f)(x) = \max\{ f(y) : y \in \mathbb{B}_x \}$$

En imágenes binarias, esta operación equivale a exigir que el conjunto reflejado $\hat{\mathbb{B}}$, trasladado a la coordenada x, tenga una **intersección no vacía** con el objeto A:

$$A \oplus \mathbb{B} = \{ z \in \mathbb{E} \mid \hat{\mathbb{B}}_z \cap A \neq \emptyset \}$$

Efecto Visual: *Expande* las estructuras claras de la imagen, aumentando el relleno de objetos, conectando componentes cercanos y eliminando canales, fosas oscuras o valles que sean geométricamente menores que el dominio $\mathbb{B}$.

4.3.1.4 Implementación de la dilatación

La versión didáctica `mm.dil0` implementa el caso particular de dilatación con **elemento estructurante plano**. Para cada píxel (y, x), la función recorre los vecinos espaciales permitidos por B y almacena el mayor valor encontrado en la imagen de entrada f, reproduciendo directamente la operación de máximo local para $b \equiv 0$.

Antes de iniciar el barrido espacial, el elemento estructurante sufre una reflexión geométrica mediante la instrucción `np.flip(Bc)` para construir explícitamente la matriz transpuesta $\hat{B}$ exigida por la teoría. En máscaras perfectamente simétricas (como cruces, cuadrados y discos centrados en el origen), esta reflexión no altera la disposición de los píxeles; no obstante, para elementos asimétricos, tal etapa es estrictamente necesaria para garantizar la equivalencia con las definiciones formales y salvaguardar las leyes de dualidad.

Así como se verificó en el operador de erosión, los valores de los vecinos siempre se leen de forma estática a partir de la matriz original f, mientras que la matriz de salida g actúa puramente como el registrador

del máximo acumulado de la vecindad. En las fronteras de la imagen, la inicialización del acumulador en 0 emula con exactitud el relleno externo por elemento neutro ($-\infty$, o cero en representaciones de 8 bits), garantizando que los bordes físicos de la escena sean dilatados en perfecta conformidad con el estándar adoptado por OpenCV.

El ejemplo computacional siguiente ilustra la aplicación práctica de un elemento en cruz (`mm.secross()`), validando la consistencia entre la lógica en bucles (`mm.dil0`) y el método nativo industrial (`mm.dil`).

```
def dil(f, Bc=np.zeros((3,3),dtype='uint8')):
    """Dilatación (OpenCV o con pesos)."""
    try:    return cv2.dilate(f, Bc)
    except: return mm.dil1(f, Bc)

def dil0(f, Bc=np.zeros((3,3),dtype='uint8')):
    """Dilatación plana siguiendo rigurosamente la teoría."""
    g = np.empty_like(f)
    Bc = np.flip(Bc)     # reflexión explícita: B
    for y in range(f.shape[0]):
        for x in range(f.shape[1]):
            g[y,x] = 0 # Inicializa con el valor mínimo para buscar el máximo
            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

# Script de Prueba y Validación
B = mm.secross()
print("Elemento estructurante B:")
print(mm.drawImage(B))

f = mm.randomImage(5,5)
print("Imagen original f:")
print(mm.drawImage(f))

print("Dilatación Plana Didáctica (dil0):")
print(mm.drawImage(dil0(f, B)))

print("Dilatación Optimizada OpenCV (dil):")
print(mm.drawImage(dil(f, B)))
```

```
Elemento estructurante B:
0 1 0
1 1 1
0 1 0

Imagen original f:
3 6 9 5 0
2 6 7 7 1
9 9 5 8 4
3 8 3 9 7
5 7 3 5 7

Dilatación Plana Didáctica (dil0):
6 9 9 9 5
9 9 9 8 7
9 9 9 9 8
9 9 9 9 9
7 8 7 9 7

Dilatación Optimizada OpenCV (dil):
```

```
6 9 9 9 5
9 9 9 8 7
9 9 9 9 8
9 9 9 9 9
7 8 7 9 7
```

Nota: El Enfrentamiento de los Signos ($f(x+z)$ vs $f(x-z)$)

Compare las definiciones formales de la erosión (Ecuación 4.3) y de la dilatación (Ecuación 4.4). Considere un elemento estructurante asimétrico a la derecha $\mathbb{B} = \{0, 1\}$ (origen y un píxel a la derecha) aplicado en la posición $x = 10$.

1. **En la Erosión** (Ecuación 4.3):

$$\min\{f(x+z) - b(z)\}$$

   - $z = 0 \Rightarrow f(10+0) = \mathbf{f(10)}$
   - $z = 1 \Rightarrow f(10+1) = \mathbf{f(11)}$

   El operador consulta el píxel actual (10) y el píxel a la derecha (11), preservando la orientación original de $\mathbb{B}$.

2. **En la Dilatación** (Ecuación 4.4):

$$\max\{f(x-z) + b(z)\}$$

   - $z = 0 \Rightarrow f(10-0) = \mathbf{f(10)}$
   - $z = 1 \Rightarrow f(10-1) = \mathbf{f(9)}$

   Debido al signo negativo ($-z$), avanzar en el elemento estructurante corresponde a retroceder en la imagen, haciendo que la dilatación consulte el píxel a la izquierda (9).

La función `_viz`, utilizada en `morph.py`, genera vecinos mediante desplazamientos aditivos de la forma $x + z$. Por ese motivo, la implementación de `mm.dil0` refleja previamente el elemento estructurante mediante `np.flip(B)`. Tras la reflexión, el barrido basado en $x + z$ pasa a acceder exactamente a los mismos puntos definidos por la expresión teórica $f(x - z)$ de la dilatación en Ecuación 4.4.
Para elementos estructurantes simétricos (como discos, cuadrados y cruces centradas), la reflexión no altera la máscara. En cambio, para elementos asimétricos, esta etapa es indispensable para que la implementación reproduzca correctamente la definición matemática de la dilatación y preserve la dualidad erosión–dilatación.

Dualidad erosión–dilatación

La erosión y la dilatación son **duales por complemento**. Esto significa que un operador puede obtenerse completamente a partir del otro, siempre que se actúe sobre el complemento de la imagen utilizando el elemento estructurante reflejado $\hat{B}$:

$$(A \ominus B)^c = A^c \oplus \hat{B} \quad \Longleftrightarrow \quad A \ominus B = (A^c \oplus \hat{B})^c$$

De manera análoga, la **dilatación también puede obtenerse a partir de la erosión**:

$$(A \oplus B)^c = A^c \ominus \hat{B} \quad \Longleftrightarrow \quad A \oplus B = (A^c \ominus \hat{B})^c$$

En términos prácticos, la erosión de un objeto puede obtenerse mediante la dilatación de su complemento, seguida de la complementación del resultado (y viceversa). En la implementación del paquete `morph.py`, las versiones didácticas `mm.ero0` y `mm.dil0` hacen explícita esta estructura mediante bucles (*loops*), mientras que `mm.ero` y `mm.dil` delegan las operaciones a OpenCV buscando una mayor eficiencia computacional.

Condiciones de contorno e imágenes finitas

En la morfología matemática clásica, definida sobre un dominio infinito (típicamente $\mathbb{Z}^2$), esta dualidad es exacta. En imágenes digitales, sin embargo, se trabaja con matrices finitas, y el resultado pasa a depender de la forma en que se tratan los píxeles ubicados fuera de la imagen.
Para que las identidades de dualidad permanezcan válidas, el complemento debe definirse con respecto al mismo universo y las condiciones de contorno adoptadas para la erosión y para la dilatación deben ser complementarias entre sí. Por ejemplo, si la erosión asume que los píxeles externos pertenecen al objeto (255), entonces la dilatación aplicada al complemento debe asumir que esos mismos píxeles externos pertenecen al fondo (0).
Cuando se utilizan diferentes estrategias de relleno (replicación, reflexión, valor constante, etc.), la dualidad teórica puede dejar de satisfacerse exactamente en las regiones cercanas a los bordes de la imagen.

Para ilustrar numéricamente los operadores morfológicos y la dualidad erosión–dilatación, la Figura 4.4 presenta una imagen binaria de 10×10 procesada con un elemento estructurante en forma de "L". En la implementación de `morph.py`, el origen de B se fija en el centro geométrico de la máscara —posición $(1,1)$ para un *kernel* de 3×3— y debe corresponder a un elemento activo para que la erosión se comporte correctamente (según lo discutido anteriormente). El elemento estructurante B_L definido a continuación satisface esa condición. La Figura 4.5 complementa el análisis con un simulador interactivo de la erosión, permitiendo visualizar el desplazamiento del elemento estructurante sobre la imagen e identificar las posiciones en las que este permanece completamente contenido en el objeto.

```
A = np.array([
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,1,1,1,0,0,0,0],
    [0,0,1,1,1,1,1,0,0,0],
    [0,1,1,1,1,1,1,1,0,0],
    [0,1,1,1,1,1,1,1,0,0],
    [0,1,1,1,1,1,1,0,0,0],
    [0,0,1,1,1,1,1,1,0,0],
    [0,0,0,1,1,1,1,0,0,0],
    [0,0,0,0,1,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0]], dtype=np.uint8) * 255

B_L   = np.array([[1,0,0],[1,1,0],[1,1,0]], dtype=np.uint8)   # 'L', origen en el centro
B_hat = np.array([[0,1,1],[0,1,1],[0,0,1]], dtype=np.uint8)   # B_L girado 180° (B)

print("Elemento estructurante B_L:"); print(mm.drawImg(B_L))
print("Elemento estructurante reflejado B:"); print(mm.drawImg(B_hat))

img_ero0 = mm.ero0(A, B_L)

# Dualidad: (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"Dualidad (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)
)
```

```
Elemento estructurante B_L:
1 0 0
```

```
1 1 0
1 1 0

Elemento estructurante reflejado B:
0 1 1
0 1 1
0 0 1

Dualidad (A   B)  == A   B : True
```

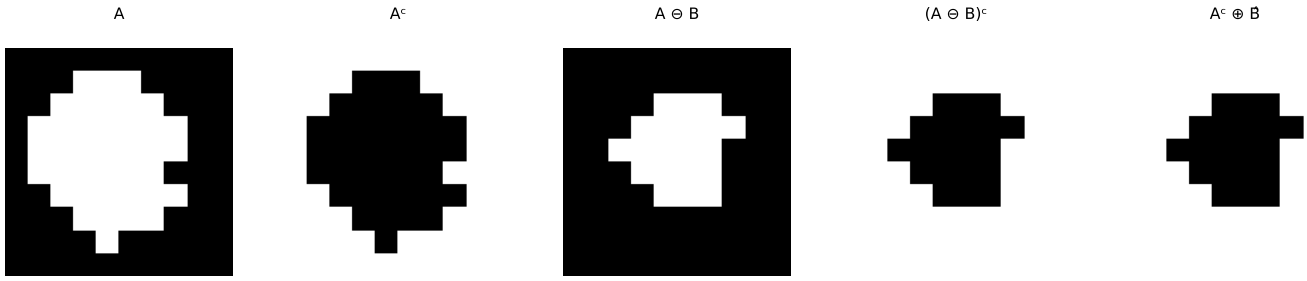
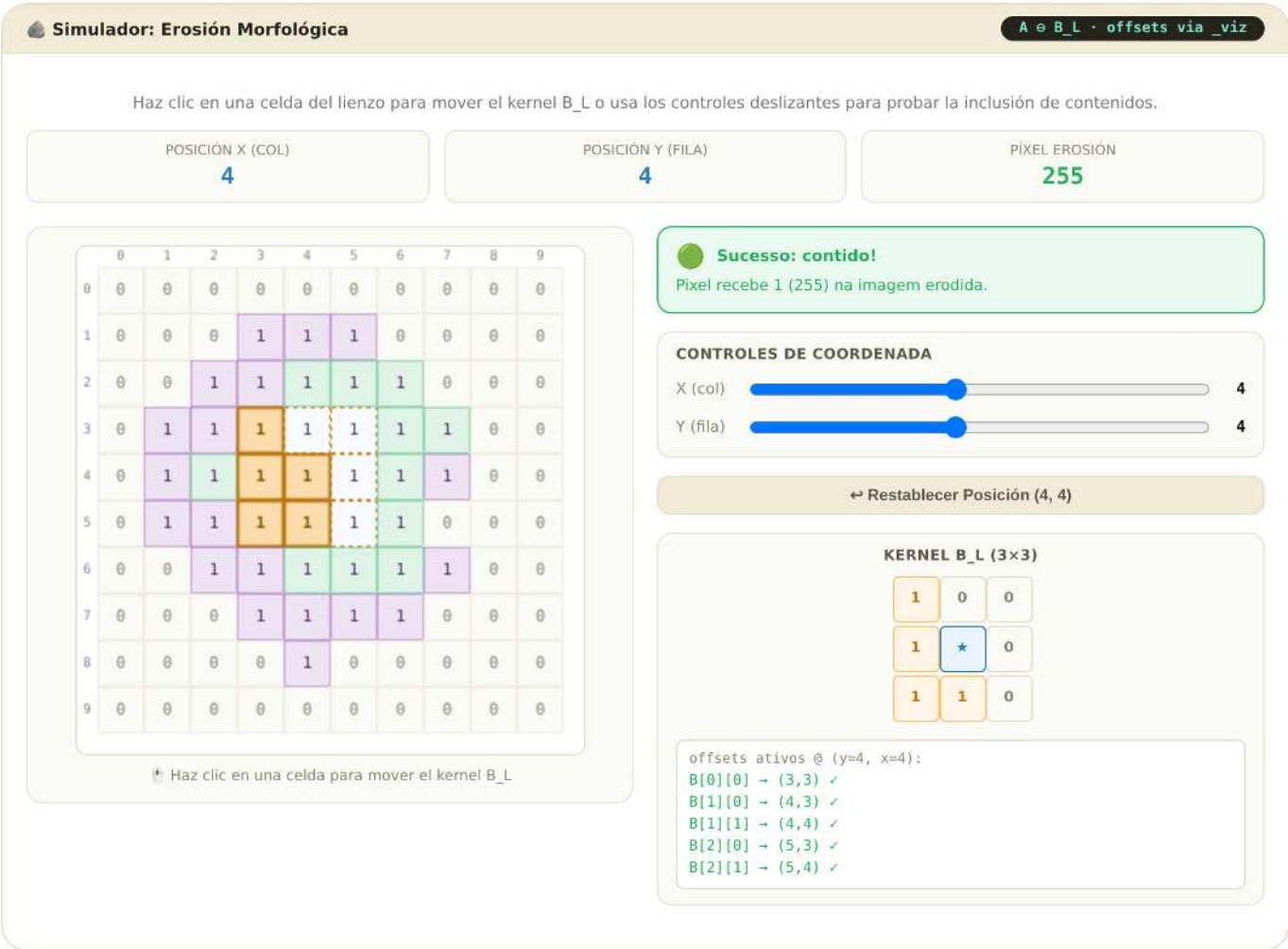


Figura 4.4: Erosión y dilatación en imagen binaria 10×10 con elemento estructurante 'L' asimétrico 3×3. Validación de la dualidad erosión–dilatación.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-04-erosao.html

Figura 4.5: Simulador: Erosión Morfológica (A B_L)

4.3.2 Apertura y Cierre

Combinando erosión y dilatación se obtienen dos operadores de gran utilidad práctica: la **apertura** y el **cierre**, definidos por las Ecuaciones Ecuación 4.5 y Ecuación 4.6. Sus principales efectos se resumen en la Tabla 4.2.

Apertura (*opening*) — erosión seguida de dilatación por el mismo B:

$$A \circ B = (A \ominus B) \oplus B \tag{4.5}$$

Cierre (*closing*) — dilatación seguida de erosión por el mismo B:

$$A \bullet B = (A \oplus B) \ominus B \tag{4.6}$$

En la práctica, `mm.open` y `mm.close` aplican el mismo elemento estructurante en las dos etapas. Para elementos estructurantes simétricos (los más comunes), esta implementación coincide con la definición matemática presentada anteriormente.

Tabla 4.2: Propiedades de apertura y cierre.

Operador	Secuencia	Efecto principal
Apertura $A \circ B$	erosión $\rightarrow$ dilatación	Elimina estructuras incapaces de contener el elemento estructurante; suaviza contornos externos
Cierre $A \bullet B$	dilatación $\rightarrow$ erosión	Rellena agujeros más pequeños que B; suaviza contornos internos

Propiedad importante: ambos son **idempotentes**. Por ejemplo,

$$(A \circ B) \circ B = A \circ B,$$

es decir, después de la primera aplicación, nuevas aplicaciones del mismo operador no alteran más el resultado.

4.3.2.1 Implementación de la apertura y el cierre

A diferencia de la erosión y la dilatación, la apertura y el cierre no introducen nuevos mecanismos computacionales. Ambos se obtienen mediante la composición secuencial de los operadores primitivos ya presentados:

```
def open0(f, B):
    return mm.dil0(mm.ero0(f, B), B)

def close0(f, B):
    return mm.ero0(mm.dil0(f, B), B)
```

La función `mm.open` delega la operación a `mm.open(f, B)`, mientras que `mm.close` utiliza `mm.close(f, B)`, produciendo el mismo resultado de forma más eficiente.

La apertura hereda de la erosión la capacidad de eliminar estructuras más pequeñas que el elemento estructurante y de la dilatación la restauración parcial de las regiones preservadas. El cierre realiza el proceso inverso: primero expande los objetos y luego restaura sus dimensiones originales, rellenando huecos y agujeros más pequeños que el elemento estructurante.

4.3.2.2 Filtro Secuencial Alternado

En la práctica, la apertura y el cierre se aplican frecuentemente en secuencia para eliminar simultáneamente el ruido externo y rellenar los huecos internos. La función `mm.asf` (*Filtro Secuencial Alternado*) generaliza esta estrategia al aplicar aperturas y cierres alternadamente con elementos estructurantes progresivamente más grandes. Las secuencias disponibles se presentan en la Tabla 4.3.

Tabla 4.3: Secuencias del filtro secuencial alternado mm.asf.

Secuencia	Orden	Uso típico
`'OC'`	apertura → cierre	elimina el ruido externo antes de rellenar pequeños huecos
`'CO'`	cierre → apertura	rellena pequeños huecos antes de eliminar el ruido externo
`'OCO'`	apertura → cierre → apertura	enfatiza la eliminación del ruido externo
`'COC'`	cierre → apertura → cierre	enfatiza el relleno de huecos y lagunas

El parámetro `n` controla el número de escalas utilizadas por el filtro. En cada iteración i, el elemento estructurante se amplía mediante la suma de Minkowski (`mm.sesum(b, i)`), produciendo una secuencia de filtros morfológicos cada vez más abarcativos. A diferencia de una única apertura o cierre con un elemento estructurante grande, el ASF realiza una suavización progresiva en múltiples escalas, preservando mejor la geometría de los objetos relevantes mientras elimina estructuras menores. La Figura 4.6 presenta un ejemplo de aplicación de la apertura, el cierre y el ASF en la imagen de las monedas.

```
img_bin    = mm.threshold(img_clahe)
B_disk     = mm.sedisk(19)

img_open  = mm.open(img_bin, B_disk)             # erosión → dilatación
img_close = mm.close(img_bin, B_disk)            # dilatación → erosión
img_oc    = mm.close(img_open, B_disk)           # apertura seguida de cierre
img_asf   = mm.asf(img_bin, 'OC', mm.sedisk(3), n=7)
# ASF: disco base 3×3, crece en cada iteración

mm.show(
    [img_bin, img_open, img_close, img_oc, img_asf],
    titles=["Binarización Otsu", "Apertura (A B)", "Cierre (A B)",
            "Apertura→Cierre", "ASF-OC (n=7)"],
    cols=5, figsize=(15, 12)
)
```

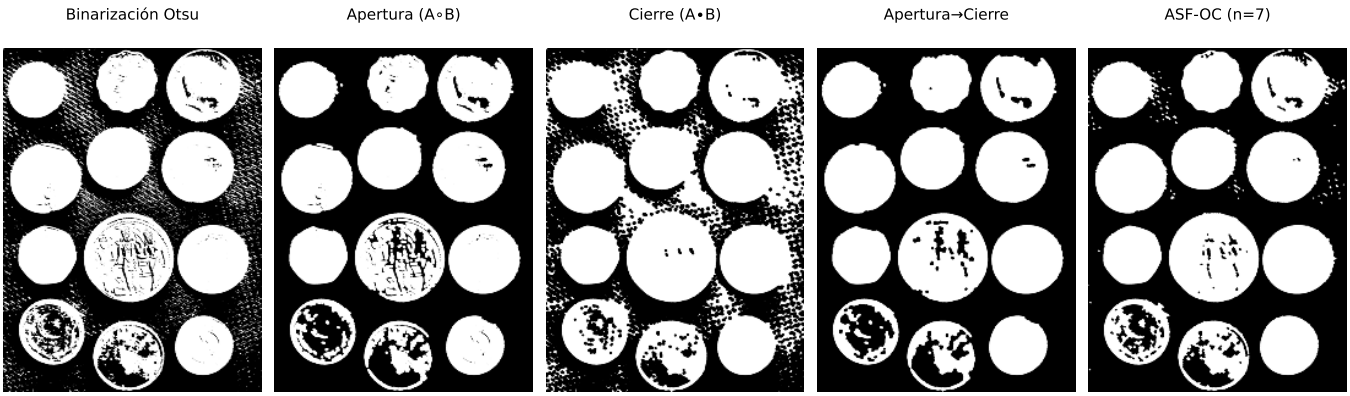


Figura 4.6: Apertura, cierre, composición y filtro secuencial alternado aplicados a la binarización Otsu de las monedas (preprocesadas por realce de contraste). Elemento estructurante: disco 13×13.

4.3.3 Operadores Geodésicos

Los operadores geodésicos introducen una restricción adicional a los operadores morfológicos clásicos mediante una imagen de control denominada **máscara** g. En lugar de permitir que la erosión o la dilatación se propaguen libremente por la imagen, el resultado de cada iteración se limita punto a punto por los valores de la máscara, restringiendo la evolución de la operación a las regiones permitidas.

4.3.3.1 Dilatación Geodésica

La **dilatación geodésica** de una imagen marcador f bajo una imagen máscara g, utilizando un elemento estructurante plano b, se define por:

$$f \oplus_g b = (f \oplus b) \wedge g, \tag{4.7}$$

donde $\wedge$ representa el mínimo punto a punto.

En otras palabras, se realiza inicialmente una dilatación convencional sobre el marcador y, a continuación, el resultado se restringe mediante la máscara g. De esta manera, la propagación nunca puede superar las regiones permitidas por la máscara.

La formulación clásica de la dilatación geodésica presupone que el marcador esté contenido en la máscara, es decir, $f \leq g$, garantizando que la evolución de la operación permanezca siempre limitada por la máscara.

4.3.3.2 Implementación de la dilatación geodésica

La función `mm.cdil` implementa directamente este operador y permite ejecutar múltiples iteraciones consecutivas:

```
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
```

La instrucción `np.minimum(mm.dil(y, b), g)` implementa exactamente la definición matemática de la dilatación geodésica, es decir, $(y \oplus b) \wedge g$.

Cuando $n = 1$, la función ejecuta una única dilatación geodésica. Para $n > 1$, el resultado de cada etapa se convierte en el marcador de la etapa siguiente, produciendo una propagación progresiva controlada por la máscara.

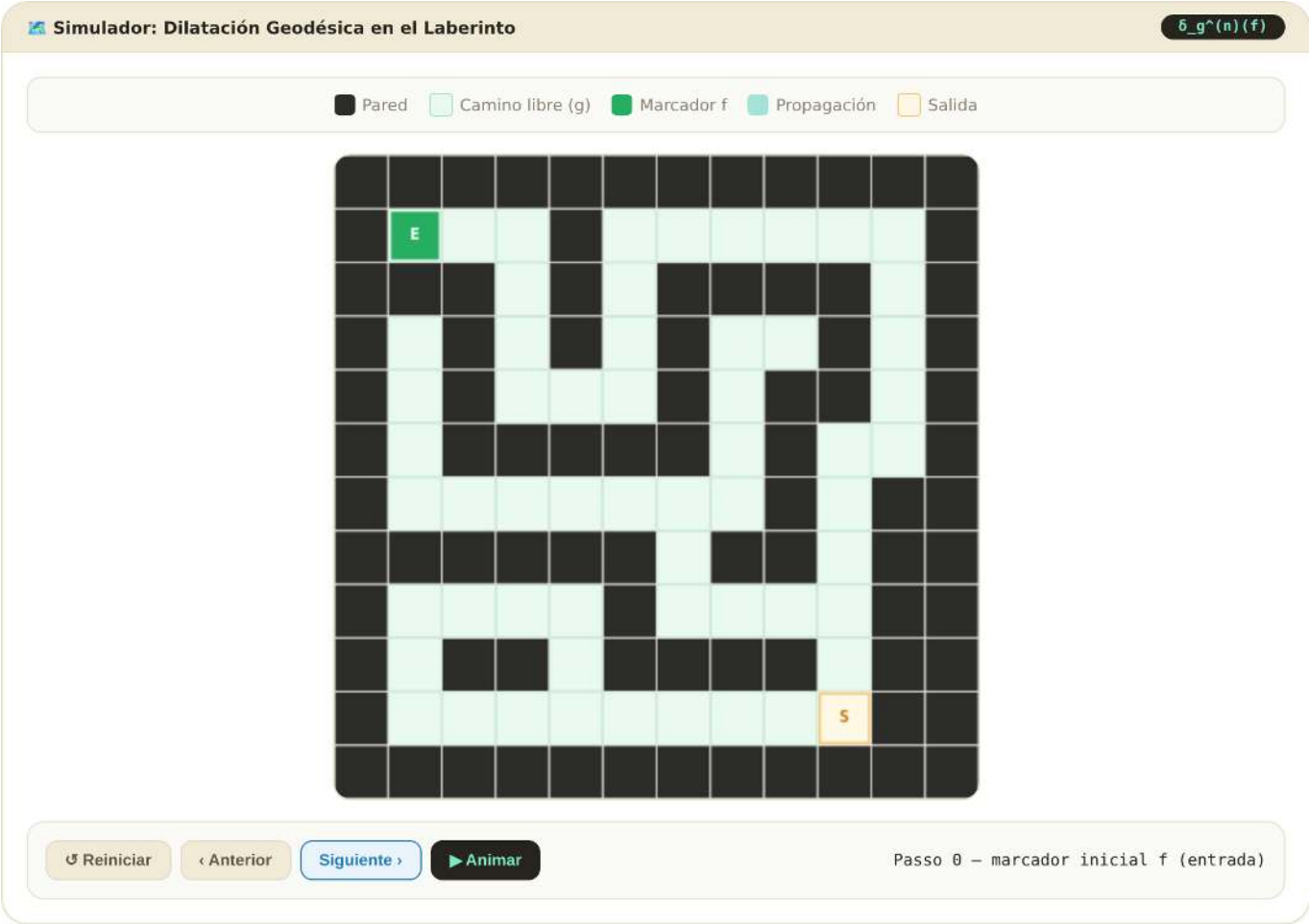
El simulador interactivo Figura 4.7 permite seguir, paso a paso, la propagación del marcador f a lo largo de los corredores del laberinto. En cada iteración de `mm.cdil`, el frente de dilatación avanza hacia las celdas vecinas libres —aquellas en las que $g = 1$—, mientras que las paredes ($g = 0$) permanecen intransitables. El número de pasos necesarios para que el marcador alcance la salida corresponde exactamente a la longitud geodésica del camino más corto dentro de la máscara, evidenciando la conexión directa entre la dilatación geodésica iterada y la noción de distancia en grafos.

4.3.3.3 Erosión Geodésica

De forma dual, la **erosión geodésica** de una imagen marcador f bajo una imagen máscara g se define por:

$$f \ominus_g b = (f \ominus b) \vee g, \tag{4.8}$$

donde $\vee$ representa el operador de máximo punto a punto.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-04-cdil.html

Figura 4.7: Simulador interactivo de dilatación geodésica: el marcador f (verde) se propaga paso a paso por los caminos libres de la máscara g, sin atravesar paredes.

En este caso, la erosión convencional del marcador es seguida por una restricción inferior impuesta por la máscara. Así, ningún píxel del resultado puede asumir un valor inferior al correspondiente píxel de la máscara.

La formulación clásica de la erosión geodésica presupone la condición dual

$$f \geq g,$$

de modo que la máscara actúe como límite inferior durante todo el proceso.

4.3.3.4 Implementación de la erosión geodésica

La función estática `mm.cero` materializa este operador:

```
staticmethod
def cero(f, g, b=np.zeros((3,3),dtype='uint8'), n=1):
    """Erosión geodésica del marcador f bajo la máscara g."""
    y = f.copy()
    for _ in range(n):
        y = np.maximum(mm.ero(y, b), g)
    return y
```

La instrucción `np.maximum(mm.ero(y, b), g)` implementa directamente la expresión $(y \ominus b) \vee g$.

Al igual que en la dilatación geodésica, el parámetro n define cuántas erosiones geodésicas sucesivas se calcularán antes de devolver la imagen final.

> **Relación con la reconstrucción morfológica**
>
> La reconstrucción morfológica presentada en la siguiente sección se obtiene mediante la aplicación iterativa de la dilatación geodésica (`mm.cdil`) hasta alcanzar un punto fijo, es decir, hasta que ninguna celda cambie de valor entre dos iteraciones consecutivas. En otras palabras, la reconstrucción consiste en una secuencia de dilataciones geodésicas sucesivas que se propagan dentro de la máscara hasta que no haya más alteraciones.
> De forma dual, también es posible definir reconstrucciones basadas en erosión geodésica mediante aplicaciones sucesivas de `mm.cero`.

4.3.3.5 Ejemplo: propagación en un laberinto mediante dualidad

La Figura 4.8 ilustra la resolución del problema de conectividad de un laberinto utilizando la dualidad morfológica por medio de las funciones `mm.cero` y `mm.suprec`.

En lugar de propagar un marcador por los corredores libres mediante dilataciones geodésicas, el problema se formula en el dominio complementario. Inicialmente, la máscara original se invierte,

$$g = 1 - g_{orig},$$

de modo que las paredes pasan a tomar el valor 1 y los corredores el valor 0. De manera análoga, el marcador se construye en ese mismo dominio complementario, conteniendo un único valor 0 en la posición de entrada del laberinto y valor 1 en los demás píxeles.

Utilizando el elemento estructurante en cruz (`mm.secross()`), la erosión geodésica actúa sobre el marcador complementado. En cada iteración de `mm.cero`, la región conectada al marcador inicial sufre erosiones sucesivas, mientras que la máscara impone un límite inferior que impide la propagación a través de las paredes del laberinto.

Las imágenes intermediarias muestran estados de la evolución después de diferentes números de iteraciones (`n=5`, `n=12` y `n=22`). El resultado final se obtiene mediante la reconstrucción geodésica por erosión (`mm.suprec`), que aplica erosiones geodésicas sucesivas hasta alcanzar un punto fijo, es decir, una situación en la que no se produce ninguna alteración adicional entre dos iteraciones consecutivas.

En el dominio complementario, la región reconstruida corresponde exactamente al conjunto de corredores conectados a la entrada del laberinto. Así, la conectividad entre la entrada y la salida puede determinarse directamente a partir de la imagen reconstruida.

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

# 1. Máscara original g (0 = Pared, 1 = Pasillo)
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)

# Inversión global al inicio (Dualidad Morfológica)
g = 1 - g_orig  # Ahora: 1 = Pared, 0 = Pasillo
f = np.ones((10, 10), dtype=np.uint8)
f[0, 1] = 0     # Semilla inyectada como 0 en el pasillo

# Elemento estructurante en cruz
B_cruz = mm.secross()

# Procesamiento directo usando mm.cero y mm.suprec en el dominio invertido
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)

# --- CONFIGURACIÓN DE LA EXHIBICIÓN GRÁFICA ---

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')

# Mapa de colores adaptado para el dominio complementario:
# En el dominio invertido: 1 = Pared (Azul Oscuro)
# Donde g == 0 e imagen == 1 = Pasillo Libre (Gris Claro)
# Donde g == 0 e imagen == 0 = Onda Geodésica Activa (Dorado)
cmap_pipeline = mcolors.ListedColormap(['#ffc13b', '#e0e0e0', '#1e3d59'])

for i, ax in enumerate(axes):
    if i == 0 or i == 1:
        # Para las condiciones iniciales (~g y ~f)
        cmap_init = mcolors.ListedColormap(['#e0e0e0', '#1e3d59'])
```

```
        ax.imshow(images[i], cmap=cmap_init, vmin=0, vmax=1)
    else:
        # Renderización basada en los estados complementarios
        render_step = np.zeros_like(g, dtype=np.uint8)
        render_step[g == 1] = 2           # Pared (1 original del mapa de colores)
        render_step[g == 0] = 1           # Pasillo estándar
        render_step[images[i] == 0] = 0   # Onda activa (0 original del mapa de colores)
        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)

    # Indicadores gráficos de Entrada y Salida
    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()
```

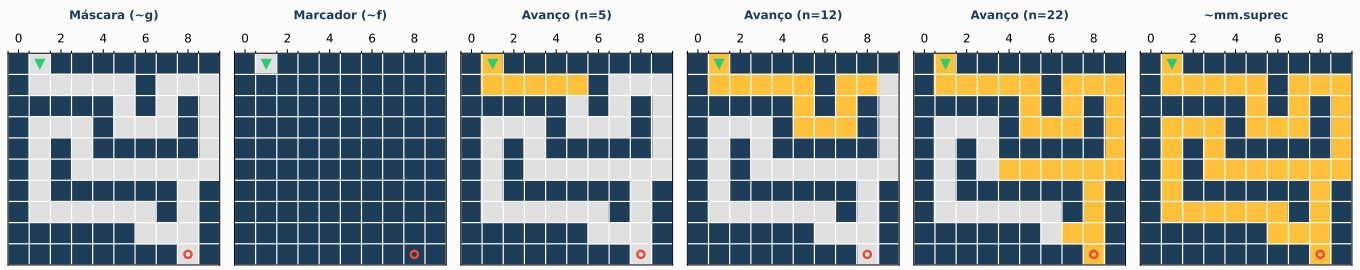


Figura 4.8: Resolución de laberinto mediante Operaciones Complementarias: Invirtiendo la máscara y el marcador al inicio del *pipeline*, el flujo se resuelve directamente en el dominio complementario a través de mm.cero y mm.suprec, eliminando reinversiones redundantes.

4.3.4 Reconstrucción Morfológica

La **reconstrucción morfológica** propaga una imagen **marcadora** f dentro de una imagen **máscara** g, garantizando que el resultado nunca supere los valores de intensidad impuestos por la máscara. El operador fundamental que posibilita esta propagación contenida es la **dilatación geodésica**, definida por la Ecuación 4.7.

La reconstrucción se obtiene mediante la aplicación iterativa de esta dilatación condicionada. Inicialmente, el marcador está limitado por la máscara para establecer el estado inicial:

$$X^{(0)} = f \wedge g,$$

y las iteraciones subsiguientes se definen de forma recursiva mediante:

$$X^{(k)} = (X^{(k-1)} \oplus b) \wedge g.$$

La secuencia crece de manera monótona hasta alcanzar un punto fijo, produciendo la **reconstrucción morfológica por dilatación** (también conocida en la literatura como *inf-reconstrucción*):

$$R_g^{\delta}(f) = \lim_{k\to\infty} X^{(k)} = X^{(k)} \quad \text{cuando} \quad X^{(k)} = X^{(k-1)}. \tag{4.9}$$

El ascenso iterativo se detiene tan pronto como se logra la estabilidad, es decir, cuando dos iteraciones consecutivas producen matrices con valores absolutamente idénticos.

4.3.4.1 Implementação da reconstrucción morfológica

La rutina didáctica `mm.infrec` implementa directamente el algoritmo iterativo de punto fijo. Inicialmente, el marcador efectivo inicial $X^{(0)}$ se determina mediante la operación `np.minimum(f, g)`. Para garantizar que el bucle de verificación ejecute la primera pasada sin disparar falsas convergencias prematuras, la variable de control de la iteración anterior (`y1`) se inicializa rellenada con un valor centinela fuera del dominio de los datos (o simplemente con una matriz que fuerce la primera ejecución).

```
def infrec(f, g, b=np.zeros((3,3), dtype='uint8')):
    """Inf-reconstrucción: dilata el marcador (f   g) hasta converger bajo la máscara g."""
    y = np.minimum(f, g)
    # Inicializa y1 con valores imposibles para forzar la entrada en el bucle
    y1 = np.full_like(f, 256, dtype=np.int16)
    while not np.array_equal(y, y1):
        y1 = y.copy()
        # Aplica la dilatación geodésica: (y   b)   g
        y = np.minimum(mm.dil(y, b), g)
    return y.astype('uint8')
```

En el interior del bucle `while`, la variable `y` almacena la estimación corriente de la reconstrucción $X^{(k)}$, mientras que `y1` preserva la imagen del estadio inmediatamente anterior $X^{(k-1)}$. La instrucción de control condicional `np.minimum(mm.dil(y, b), g)` traduce fielmente la dilatación geodésica teórica, donde la expansión morfológica convencional comandada por OpenCV se "poda" inmediatamente y queda limitada por las barreras de intensidad de la máscara g. El bucle cesa cuando no se registra ninguna modificación de píxel entre los pasos.

4.3.4.2 Ventajas de la reconstrucción morfológica

La reconstrucción morfológica es significativamente más robusta que la apertura convencional porque es capaz de eliminar estructuras no deseadas sin distorsionar ni alterar la morfología de los objetos que deben conservarse.

Mientras que la apertura clásica suaviza esquinas, elimina puntas y deforma contornos debido a la imposición geométrica rígida del elemento estructurante, la reconstrucción geodésica utiliza la máscara para recuperar con exactitud los límites y formatos originales de los objetos que poseen conectividad con el marcador original.

En términos intuitivos, el marcador actúa como una semilla de contagio que se expande progresivamente, pero solo transitando por las regiones permitidas por la máscara. Los componentes que no poseen ninguna intersección con el marcador jamás serán reconstruidos (siendo eliminados), mientras que los componentes tocados por la semilla se expanden hasta restaurar integralmente su geometría original.

Este comportamiento discriminatorio y conservativo se ilustra en la Figura 4.9, utilizando el elemento estructurante en cruz presentado en la Figura 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())          # marcador: núcleo del objeto principal

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"  Estabilidad en la iteración 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)
)
```

```
Estabilidad en la iteración 5: True
```

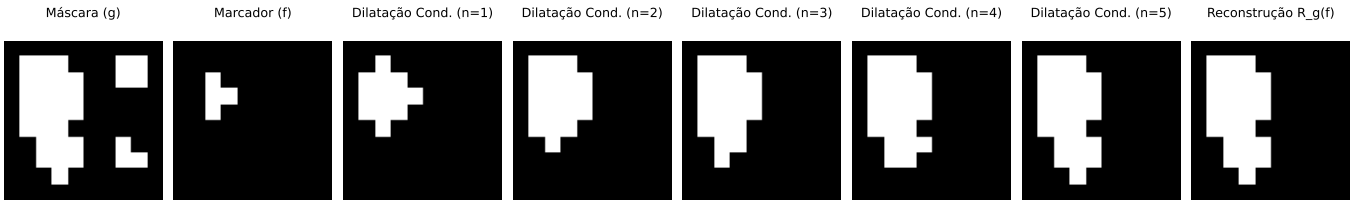


Figura 4.9: *Pipeline* de Reconstrucción Morfológica por Dilatación Condicionada: la máscara contiene dos objetos, el marcador aísla solo el núcleo del objeto principal, y las iteraciones reconstruyen su forma exacta hasta la convergencia.

4.3.5 Relleno de Agujeros y Eliminación de Bordes

Dos operadores basados en reconstrucción morfológica completan el *pipeline* de limpieza binaria. Sus características principales se resumen en la Tabla 4.4.

Relleno de agujeros (`mm.clohole`) elimina cavidades completamente rodeadas por el objeto, independientemente del tamaño, sin alterar los contornos externos. El procedimiento actúa sobre el complemento de la imagen, utilizando como marcador una restricción del marco (*frame*) al fondo:

$$\text{clohole}(f) = \left(R^{\delta}_{f^c}(\text{frame}(f) \wedge f^c)\right)^c \tag{4.10}$$

En términos operativos, primero se reconstruye el fondo externo y, a continuación, se aplica la complementación para recuperar los objetos con los agujeros rellenos.

Eliminación de objetos de borde (`mm.edgeoff`) elimina todos los objetos que tocan el borde de la imagen, preservando únicamente los componentes totalmente internos. El marcador se obtiene mediante la intersección entre el marco (*frame*) y los objetos de la imagen:

$$\text{edgeoff}(f) = f \setminus R^{\delta}_{f}(\text{frame}(f) \wedge f) \tag{4.11}$$

Tabla 4.4: Comparación entre los operadores *clohole* y *edgeoff*.

Operador	Marcador	Máscara	Efecto
`mm.clohole`	*frame* restringido al fondo (f^c)	f^c	Rellena agujeros internos
`mm.edgeoff`	*frame* restringido al objeto (f)	f	Elimina objetos conectados al borde

La evolución paso a paso de estas transformaciones geodésicas puede seguirse en las figuras siguientes. La Figura 4.10 ilustra el mecanismo de inundación controlada del operador `mm.clohole`, en el cual la reconstrucción se produce a partir del fondo externo e impide la propagación hacia regiones internas no conectadas al exterior, dando como resultado el relleno consistente de las cavidades internas. En cambio, la Figura 4.11 detalla la dinámica del operador `mm.edgeoff`, en la que solo los componentes conectados al borde se reconstruyen y posteriormente se eliminan, preservando exclusivamente los objetos totalmente contenidos en el interior de la imagen.

La conectividad de la propagación geodésica se controla mediante el elemento estructurante: `mm.sebox()` (vecindad de 8) incluye conexiones diagonales, mientras que `mm.secross()` (vecindad de 4) las excluye. En consecuencia, la elección del elemento estructurante afecta a qué componentes se alcanzan mediante la reconstrucción y, por tanto, cuáles se preservarán o eliminarán.

4.3.5.1 Conformidad con la implementación

Las definiciones anteriores están directamente alineadas con la implementación en `morph.py`, reproducida a continuación:

```
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))
```

Estas implementaciones dejan explícito que ambos operadores son instancias directas de reconstrucción morfológica por dilatación geodésica con `mm.infrec`, diferenciándose únicamente en la elección del marcador y de la máscara: `clohole` actúa sobre el complemento de la imagen, mientras que `edgeoff` actúa directamente en el dominio de los objetos.

```
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)       # borde externo

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"  Validación clohole: {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
)
```

```
  Validación clohole: True
```

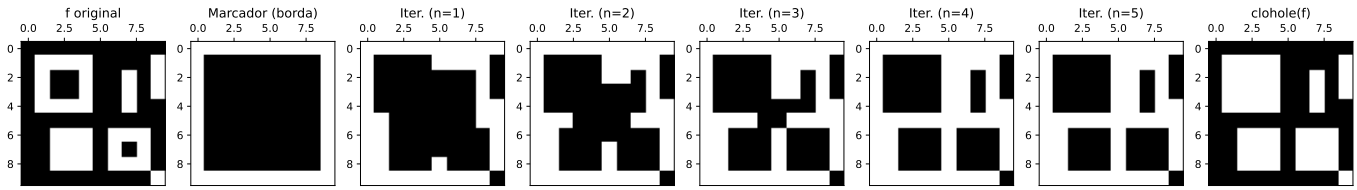


Figura 4.10: *Pipeline* de relleno de agujeros (*clohole*): el marcador proviene del borde de la imagen, restringido al complemento f^c. La dilatación geodésica reconstruye el fondo externo; tras la complementación, los agujeros internos quedan rellenos.

```
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)   # borda   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)
```

```
)
```

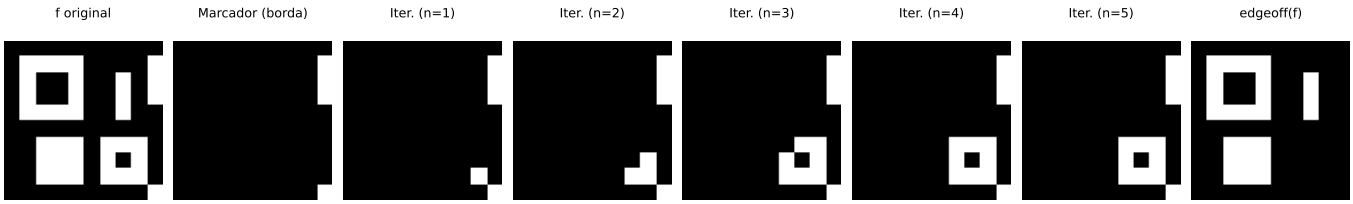


Figura 4.11: *Pipeline* de eliminación de estructuras de borde (*edgeoff*): el marcador captura las raíces conectadas a los extremos, la reconstrucción delimita esos elementos y la resta preserva solo los objetos totalmente internos.

4.3.6 *Pipeline* de Limpieza Binaria con CLAHE

Con base en el análisis anterior —en el cual el CLAHE produjo el mayor valor de la varianza interclases ($\sigma_B^2 \approx 2{,}47 \times 10^3$), ver Figura 4.2, y el operador `mm.clohole` mostró ser eficaz en el relleno de las cavidades internas—, el *pipeline* final de segmentación, ilustrado en la Figura 4.12, se estructura mediante el siguiente flujo computacional:

$$\text{gray} \xrightarrow{\text{CLAHE}} \xrightarrow{\text{Otsu}} \xrightarrow{\text{open}} \xrightarrow{\text{clohole}} \xrightarrow{\text{open}} \xrightarrow{\text{edgeoff}} \text{segmentación}$$

Tras la etapa de `mm.clohole`, se aplica una segunda apertura morfológica con un elemento estructurante mayor (`mm.sedisk(33)`, disco de diámetro 33). Esta operación elimina pequeñas regiones residuales y artefactos que podrían haber permanecido después de la segmentación. En particular, el relleno geodésico puede transformar pequeñas cavidades aisladas en componentes conectados al objeto, haciendo conveniente una etapa adicional de filtrado basada en tamaño. El diámetro se eligió de modo que las monedas sigan siendo capaces de contener el elemento estructurante, mientras que los componentes significativamente más pequeños sean eliminados.

En esta imagen, ninguna moneda está conectada al borde de la matriz. En consecuencia, la aplicación de `mm.edgeoff` no altera el resultado obtenido tras la segunda apertura. Aun así, esta etapa se mantiene en el *pipeline* por robustez, pues en otras imágenes pueden existir objetos parcialmente visibles o conectados a los bordes, que deben eliminarse antes de la etapa de análisis.

¿Por qué la apertura después del clohole?

El operador `mm.clohole` rellena todas las cavidades cerradas presentes en los objetos segmentados. En algunas situaciones, pequeñas regiones no deseadas pueden permanecer después de esta etapa o volverse conectadas a los objetos principales. La apertura morfológica subsiguiente elimina componentes más pequeños que el elemento estructurante, preservando las monedas debido a su tamaño significativamente mayor.

```
#   Preprocesamiento: solo CLAHE
img_clahe0 = mm.clahe(img_coins_gray, 2.0, 8)

#   Etapa 1: Binarización Otsu
img_bin = mm.threshold(img_clahe0)

#   Etapa 2: Apertura - elimina ruidos blancos de fondo
img_open = mm.open(img_bin, mm.sedisk(9))

#   Etapa 3: clohole - cierra todos los agujeros internos
img_hole = mm.clohole(img_open)

#   Etapa 4: Apertura (kernel grande) - elimina artefactos del clohole
```

```
img_limpo = mm.open(img_hole, mm.sedisk(33))

#   Etapa 5: edgeoff - elimina objetos que tocan el borde
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",                "Apertura (r=9)",
            "clohole",          "Apertura (r=33)",     "Final (edgeoff)"],
    cols=6, figsize=(18, 6)
)
```

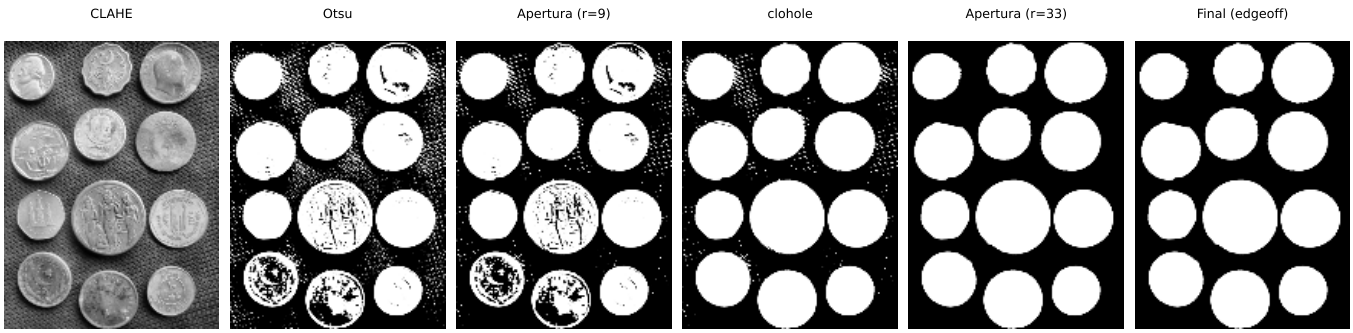


Figura 4.12: *Pipeline* completo de segmentación con CLAHE: Otsu → apertura (r=9) → clohole → apertura (r=33) → edgeoff.

4.3.7 Morfología en Tonos de Gris

Los operadores morfológicos se extienden naturalmente a imágenes en tonos de gris. En esta formulación, la erosión y la dilatación pasan a actuar directamente sobre los niveles de intensidad de la imagen. Para elementos estructurantes planos ($b \equiv 0$), la erosión corresponde al **mínimo local** y la dilatación al **máximo local** dentro de la vecindad definida por el elemento estructurante.

La interpretación intuitiva es sencilla: la erosión **oscurece** regiones al reemplazar cada píxel por el menor valor presente en su vecindad, mientras que la dilatación **aclara** regiones al utilizar el mayor valor disponible. La combinación de estos operadores permite construir transformaciones capaces de resaltar bordes, eliminar tendencias de iluminación y destacar estructuras locales.

Tres operadores derivados son especialmente útiles:

Gradiente morfológico — resalta bordes como la diferencia entre dilatación y erosión:

$$\text{grad}_B(f) = (f \oplus B) - (f \ominus B) \tag{4.12}$$

Top-hat — resalta estructuras brillantes más pequeñas que el elemento estructurante (diferencia entre la imagen original y su apertura):

$$\text{top-hat}_B(f) = f - (f \circ B) \tag{4.13}$$

Black-hat — resalta estructuras oscuras más pequeñas que el elemento estructurante (diferencia entre el cierre y la imagen original):

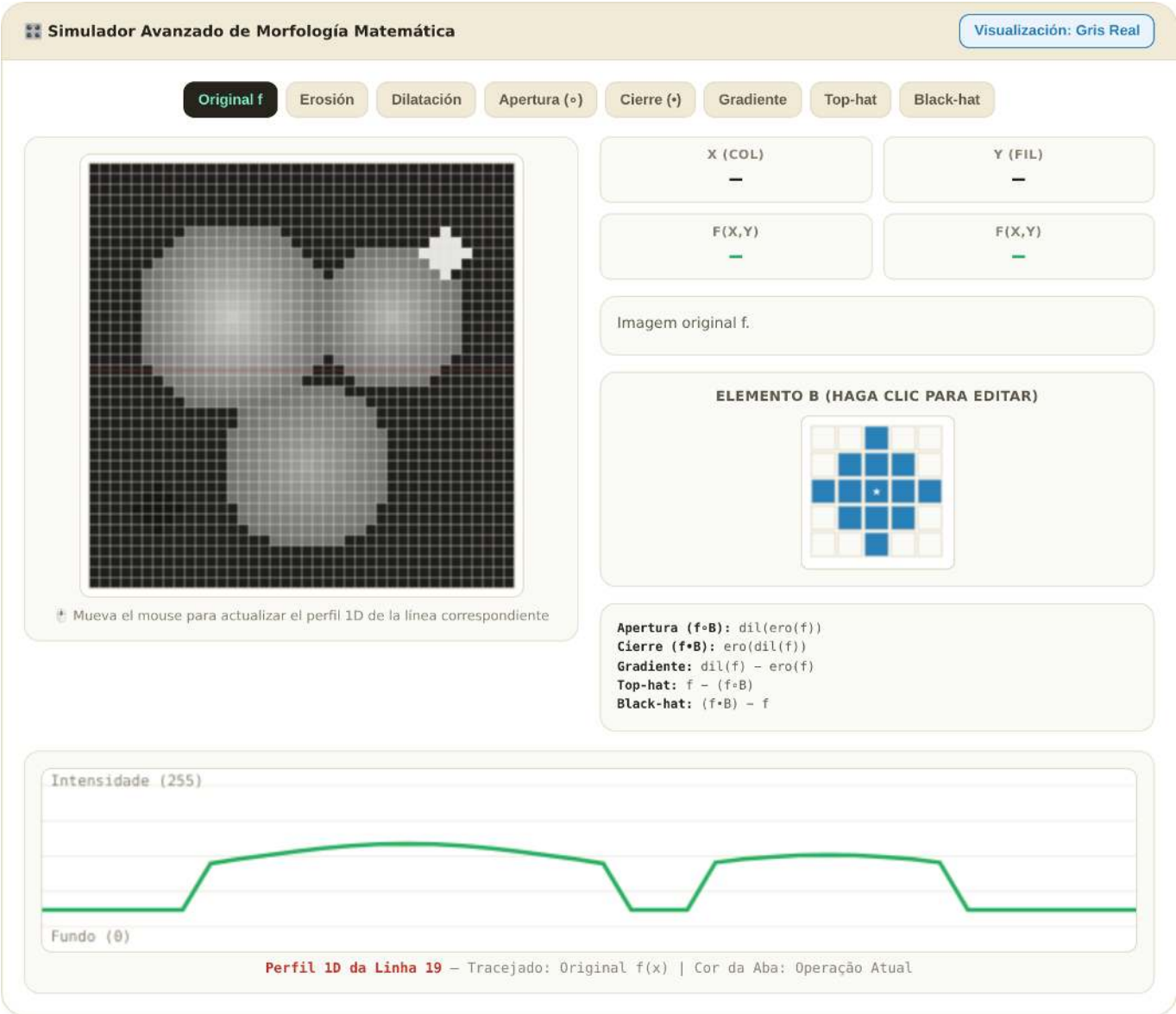
$$\text{black-hat}_B(f) = (f \bullet B) - f \tag{4.14}$$

El *Top-hat* extrae detalles brillantes que no sobreviven a la apertura, mientras que el *Black-hat* evidencia detalles oscuros eliminados por el cierre. Por su parte, el gradiente morfológico resalta transiciones abruptas de intensidad, produciendo una representación similar a la de un detector de bordes.

Para comprender el mecanismo de estos operadores a nivel local, la Figura 4.13 presenta un simulador interactivo de morfología en tonos de gris. El simulador permite editar libremente el elemento estructurante, visualizar su desplazamiento sobre la imagen y acompañar simultáneamente el perfil unidimensional de las intensidades. De esta forma, se hace posible observar directamente cómo la erosión selecciona mínimos locales, cómo la dilatación selecciona máximos locales y cómo el gradiente morfológico emerge de la diferencia entre estos dos operadores.

El botón ubicado en la esquina superior derecha permite alternar entre la visualización original en tonos de gris y una representación pseudocoloreada (*colormap*) solo en los tres tipos de gradientes. La versión coloreada facilita la percepción visual de las variaciones de intensidad, haciendo más evidente la acción de los operadores morfológicos sobre máximos, mínimos y transiciones locales de la imagen.

Los operadores presentados están disponibles en `morph.py` mediante las funciones `mm.gradm`, `mm.tophat` y `mm.blackhat`.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-04-operadores-av.html

Figura 4.13: Simulador interactivo avanzado de morfología con elemento estructurante editable y perfil 1D.

La Figura 4.14 ilustra los efectos de estos operadores sobre la imagen de las monedas y sus respectivos

histogramas. Observe que la erosión desplaza la distribución hacia intensidades más bajas, mientras que la dilatación la desplaza hacia intensidades más altas. El gradiente concentra valores en las regiones de contorno, y los operadores *Top-hat* y *Black-hat* producen histogramas fuertemente concentrados en bajos niveles de intensidad, ya que solo se resaltan pequeñas estructuras locales.

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

def fig2img(fig):
    b = io.BytesIO(); fig.savefig(b, format='png', dpi=100); plt.close(fig); b.seek(0)
    return (plt.imread(b)[:, :, :3] * 255).astype(np.uint8)

def plot_hist(img, title):
    fig, ax = plt.subplots(figsize=(4, 3))
    h = mm.hist(img)
    # CORRECCIÓN: range usa len(h) dinámicamente para coincidir con el retorno de la
    biblioteca
    ax.bar(range(len(h)), h, color='steelblue', width=1, edgecolor='steelblue')
    ax.set(xlim=(0, 255)); plt.tight_layout()
    return fig2img(fig)

# 1. Procesamiento morfológico base
B = mm.sedisk(19)
operadores = [
    ("Original", img_coins_gray),
    ("Erosão", mm.ero(img_coins_gray, B)),
    ("Dilatação", mm.dil(img_coins_gray, B)),
    ("Gradiente Morf.", mm.gradm(img_coins_gray, B)),
    ("Top-hat", mm.tophat(img_coins_gray, B)),
    ("Black-hat", mm.blackhat(img_coins_gray, B))
]

# 2. Montaje dinámico del par: [Imagen, Histograma]
imgs, titles = [], []
for nome, img in operadores:
    imgs += [img, plot_hist(img, f"Hist. {nome}")]
    titles += [nome, f"Hist. {nome}"]

# 3. Exhibición final en cuadrícula de dos columnas (Imagen | Histograma)
mm.show(imgs, titles=titles, cols=4, figsize=(10, 8))
```

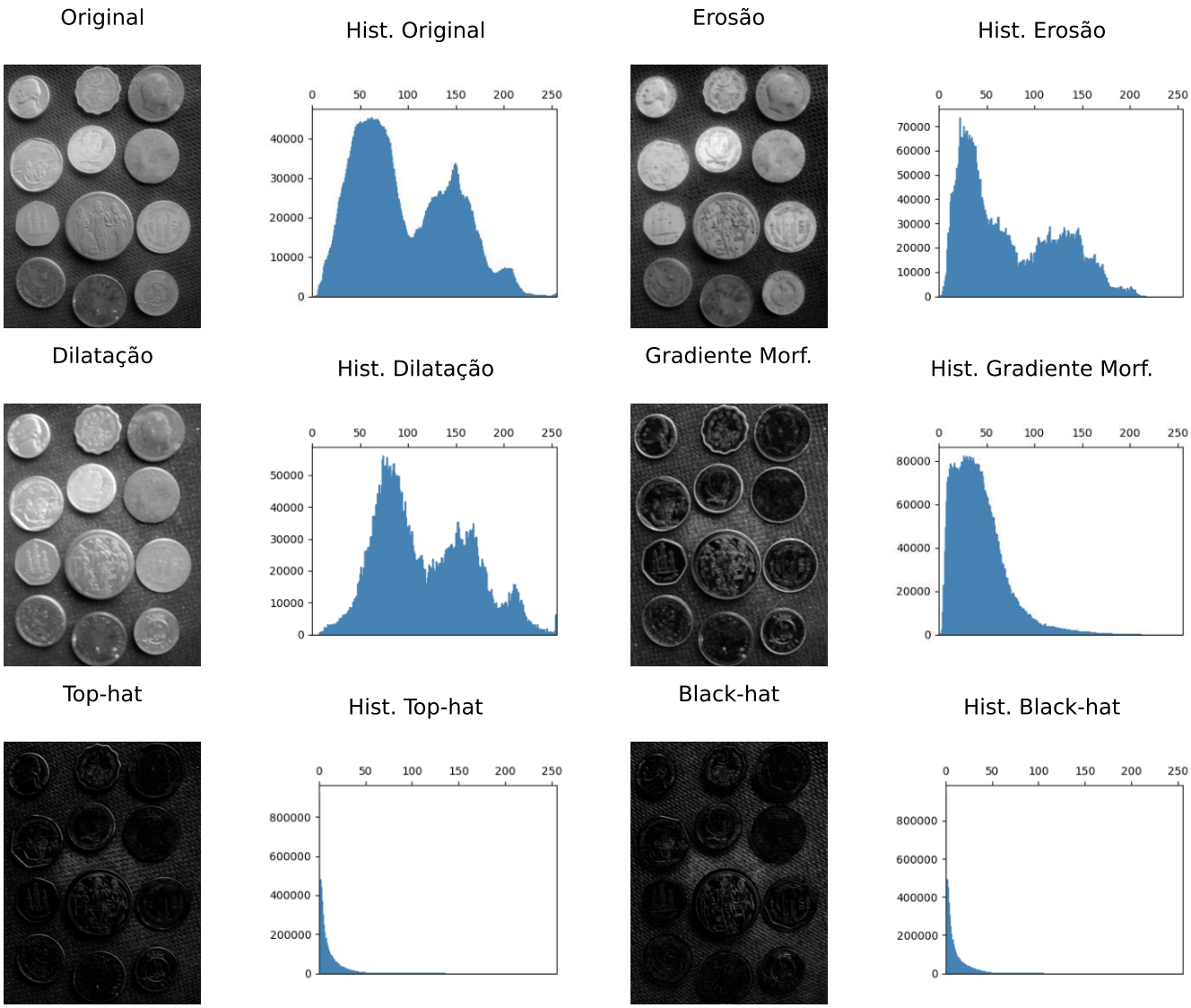


Figura 4.14: Morfología en tonos de gris y sus respectivos histogramas. La erosión reduce intensidades locales, la dilatación las amplía, el gradiente resalta bordes, y los operadores Top-hat y Black-hat evidencian detalles locales brillantes y oscuros. Elemento estructurante: disco de diámetro 19.

Los operadores morfológicos presentados anteriormente se utilizarán ahora como herramientas de refinamiento y generación de marcadores para métodos de segmentación más avanzados, que se presentan a continuación.

4.4 Segmentación de Imágenes: Fundamentación y Taxonomía

La **segmentación de imágenes** consiste en dividir la imagen en regiones asociadas a objetos o estructuras de interés. En PDI, representa la transición entre el procesamiento de bajo nivel —como filtrado y realce— y etapas de análisis más avanzadas, como extracción de características, reconocimiento e interpretación de la escena.

Formalmente, el objetivo de la segmentación consiste en descomponer el dominio espacial completo de una imagen, denotado por Ω, en una partición de subconjuntos $\{R_1, R_2, \dots, R_n\}$ que satisfaga simultáneamente los criterios de **completitud** y **disyunción**:

$$\bigcup_{i=1}^{n} R_i = \Omega, \qquad R_i \cap R_j = \emptyset \quad \forall\, i \neq j \tag{4.15}$$

Además de las propiedades de completitud y disyunción expresadas en la Ecuación 4.15, cada subregión R_i debe constituir un dominio **homogéneo** según un predicado de similitud definido sobre propiedades locales —intensidad, color o textura— y, simultáneamente, ser **distinta** de las regiones adyacentes.

Las técnicas de segmentación pueden organizarse en diferentes familias. En este capítulo se enfatizarán los enfoques resumidos en la Tabla 4.5, fundamentados principalmente en criterios de intensidad, conectividad y proximidad espacial.

Tabla 4.5: Taxonomía simplificada de los principales enfoques de segmentación y refinamiento estudiados en este capítulo.

Enfoque	Criterio de Segmentación	Operadores de Referencia
Umbralización	Particionamiento del espacio de intensidades	Criterio de Otsu, umbralización global y local
Morfología Matemática	Relaciones espaciales definidas por elementos/funciones estructurantes	Erosión, dilatación, apertura, cierre y reconstrucción
Basada en Regiones	Homogeneidad local y conectividad espacial	Etiquetado de componentes conexos, Transformada de Distancia y *Watershed*

Hasta este punto, el desarrollo práctico se ha centrado en la **umbralización**, mediante la combinación entre ecualización adaptativa CLAHE y el método global de Otsu. Esta etapa se complementó con operadores de **reconstrucción morfológica** basados en dilataciones geodésicas, implementados por las funciones `mm.infrec`, `mm.clohole` y `mm.edgeoff`, produciendo una máscara binaria limpia y adecuada para el análisis.

Sin embargo, en escenarios donde objetos distintos aparecen conectados en la máscara binaria —ya sea por contacto físico, superposición parcial o por puentes estrechos de píxeles producidos por la segmentación—, la umbralización deja de ser suficiente para individualizar cada objeto. En estos casos, múltiples objetos pasan a componer un único componente conexo, dificultando etapas posteriores de medición e interpretación.

Para superar esta limitación, las próximas secciones introducen tres herramientas complementarias: el **Etiquetado de Componentes Conexos**, la **Transformada de Distancia** y el algoritmo de segmentación por ***Watershed*** basado en marcadores. En conjunto, estas técnicas permiten separar objetos adyacentes, identificar regiones individualmente y extraer descriptores geométricos consistentes para el análisis cuantitativo.

4.4.1 Etiquetado

El **etiquetado de componentes conexas** (*connected component labeling*) es el operador que asigna un identificador entero único a cada conjunto de píxeles pertenecientes a la misma componente conexa en una imagen binaria.

> **Definición formal**
>
> Dada una imagen binaria f y una relación de conectividad definida por un elemento estructurante B (típicamente conectividad-4 o conectividad-8), el algoritmo de etiquetado de la Figura 4.15 produce una imagen g en la cual todos los píxeles pertenecientes a la misma componente conexa reciben la misma etiqueta entera positiva, mientras que los píxeles pertenecientes a componentes distintas reciben etiquetas diferentes.

La conectividad define qué píxeles se consideran vecinos directos de un píxel (x, y). Las definiciones más utilizadas son:

- **Conectividad-4:** considera solo los cuatro vecinos ortogonales (norte, sur, este y oeste).

- **Conectividad-8:** considera los cuatro vecinos ortogonales y los cuatro diagonales, totalizando ocho vecinos.

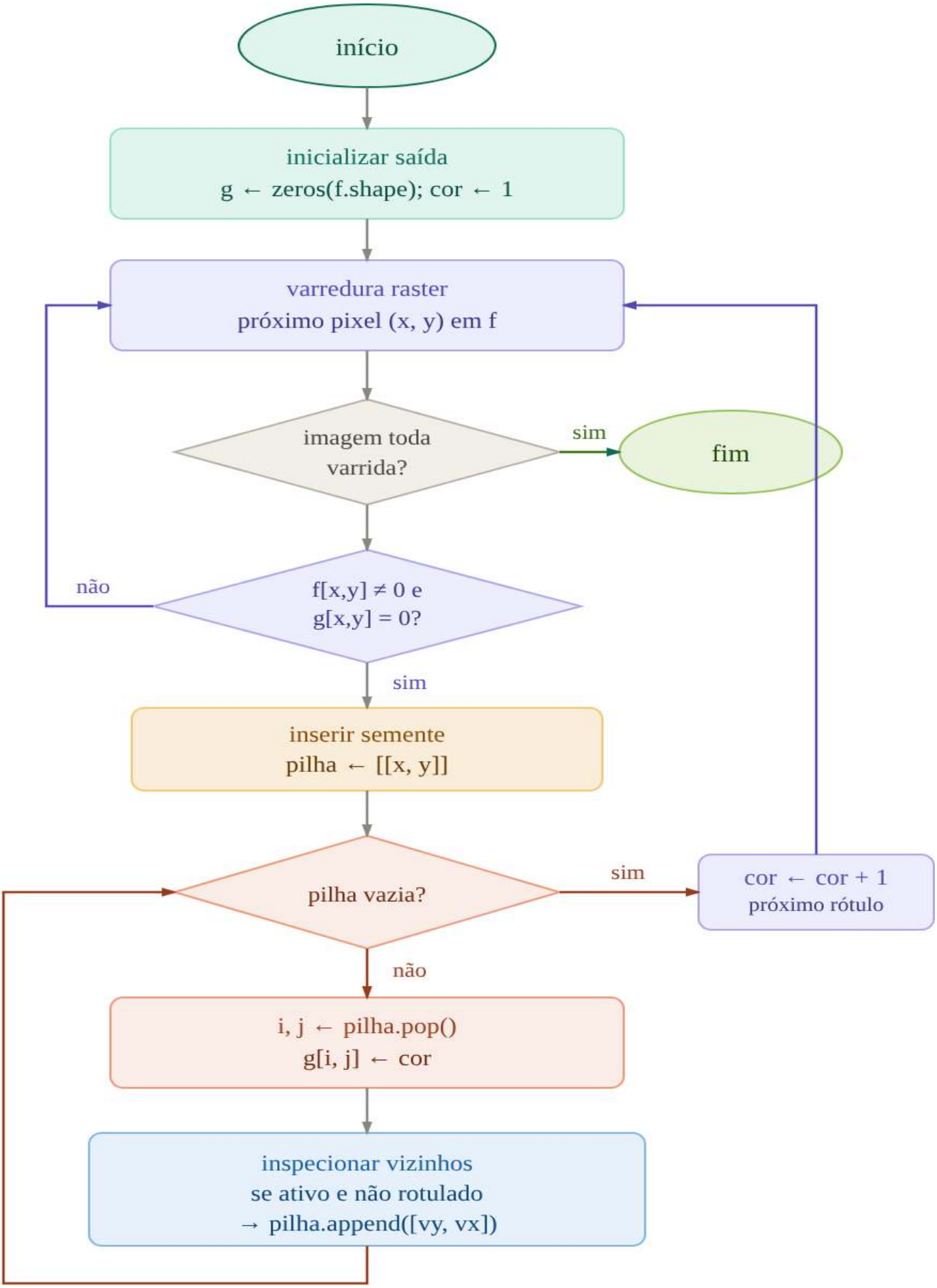
La elección de la conectividad influye directamente en la formación de las componentes conexas y, consecuentemente, en el resultado del etiquetado, como se ilustra en la Figura 4.17. Un ejemplo adicional puede explorarse de forma interactiva en el simulador presentado en la Figura 4.16.

La implementación en `morph.py` proporciona dos versiones de este operador. La función `mm.label0` reproduce explícitamente el algoritmo de *flood-fill* utilizando una pila y permite controlar la conectividad mediante el elemento estructurante adoptado. En cambio, `mm.label` delega la operación a la implementación optimizada de OpenCV (`mm.label0`). En ambos casos, el resultado es una imagen etiquetada en la cual cada componente conexa recibe un identificador entero distinto.

El auxiliar `_viz` itera sobre la ventana estructurante `b` centrada en (i, j), generando únicamente los vecinos válidos dentro de los límites de la imagen — la conectividad deseada está enteramente determinada por la forma de `b` pasada al algoritmo.

Ejemplo didáctico — efecto de la conectividad:

```
# Imagen binaria 10×10 con componentes diagonalmente adyacentes
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 con línea anterior
    [0,0,0,1,0,0,1,1,0,0],
    [0,0,0,0,0,0,1,1,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,1,1,1,0,0,0,0,0,0],
    [0,1,0,1,0,0,0,1,0,0],
    [0,1,1,1,0,0,0,0,1,0],  # diagonal con línea anterior
    [0,0,0,0,0,0,0,0,0,0]], dtype=np.uint8) * 255

# Elemento estructurante cruz (conectividad-4) y cuadrado (conectividad-8)
B4 = mm.secross()   # conectividad-4
B8 = mm.sebox()     # conectividad-8

# Etiquetación con label0 (didáctico) y 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)

# Validación cruzada
print(f"  label0 (C4) == cv2 (C4): {np.array_equal(lbl4_didatico, lbl4_cv2)}")
print(f"  label0 (C8) == cv2 (C8): {np.array_equal(lbl8_didatico, lbl8_cv2)}")
print(f"   Componentes C4: {lbl4_cv2.max()}  |  Componentes C8: {lbl8_cv2.max()}")

# Normalización para visualización
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"Etiquetación C4\n({lbl4_cv2.max()} componentes)",
        f"Etiquetación C8\n({lbl8_cv2.max()} componentes)"
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-alg-rotulagem2.html

Figura 4.15: Algoritmo de etiquetado por *flood-fill* con pila.

Versión interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-04-rotulacao.html

Figura 4.16: Simulador interactivo de etiquetado de componentes conexas (*connected component labeling*): visualización de la expansión flood-fill, conectividad-4 y conectividad-8.

```
    ],
    cols=3, figsize=(12, 4), axis=True
)
```

```
label0 (C4) == cv2 (C4): True
label0 (C8) == cv2 (C8): True
  Componentes C4: 7  |  Componentes C8: 4
```

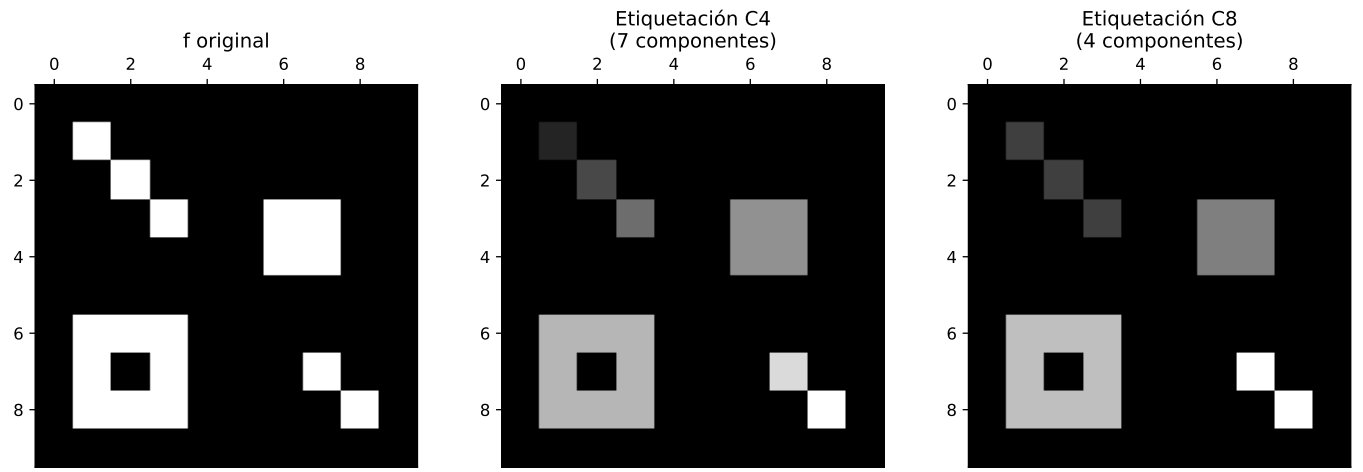


Figura 4.17: Efecto de la conectividad en la etiquetación: la imagen binaria 10×10 contiene píxeles diagonalmente adyacentes. Con conectividad-4, estos píxeles forman componentes distintas; con conectividad-8, se fusionan en un único componente.

4.4.2 Transformada de Distancia

La **Transformada de Distancia** (TD) es un operador que, aplicado a una imagen binaria f, produce una imagen en niveles de gris D en la cual cada píxel perteneciente al objeto ($f(x, y) \neq 0$) recibe como valor la distancia geométrica hasta el píxel de fondo ($f(x', y') = 0$) más cercano:

$$D(x, y) = \min_{(x',y') \,:\, f(x',y')=0} d\big((x, y),\, (x', y')\big)$$

donde $d(\cdot, \cdot)$ es una métrica de distancia — típicamente la distancia Euclidiana (L_2). El resultado es una representación topográfica de los objetos: los píxeles ubicados en el interior asumen valores elevados, mientras que los píxeles próximos a los bordes presentan bajos valores de distancia. Los **máximos locales** de D corresponden a los puntos más alejados del borde del objeto, frecuentemente próximos a sus centros geométricos o centros de máxima inscripción — propiedad particularmente útil para la generación automática de marcadores en el algoritmo *watershed*.

> **Definición formal mediante erosiones**
>
> La TD admite además una interpretación morfológica iterativa, conforme al algoritmo de la Figura 4.18. Considérese una función estructurante b cuyo valor central es nulo y cuyos vecinos poseen costos negativos asociados al desplazamiento. Al aplicar erosiones sucesivas con esta función estructurante particular, los valores de los píxeles de los objetos (que deben asumir la distancia máxima posible de la imagen) se reducen progresivamente según los costos definidos por b. El valor acumulado de esta propagación pasa entonces a representar la distancia al fondo según la métrica inducida por la función estructurante.

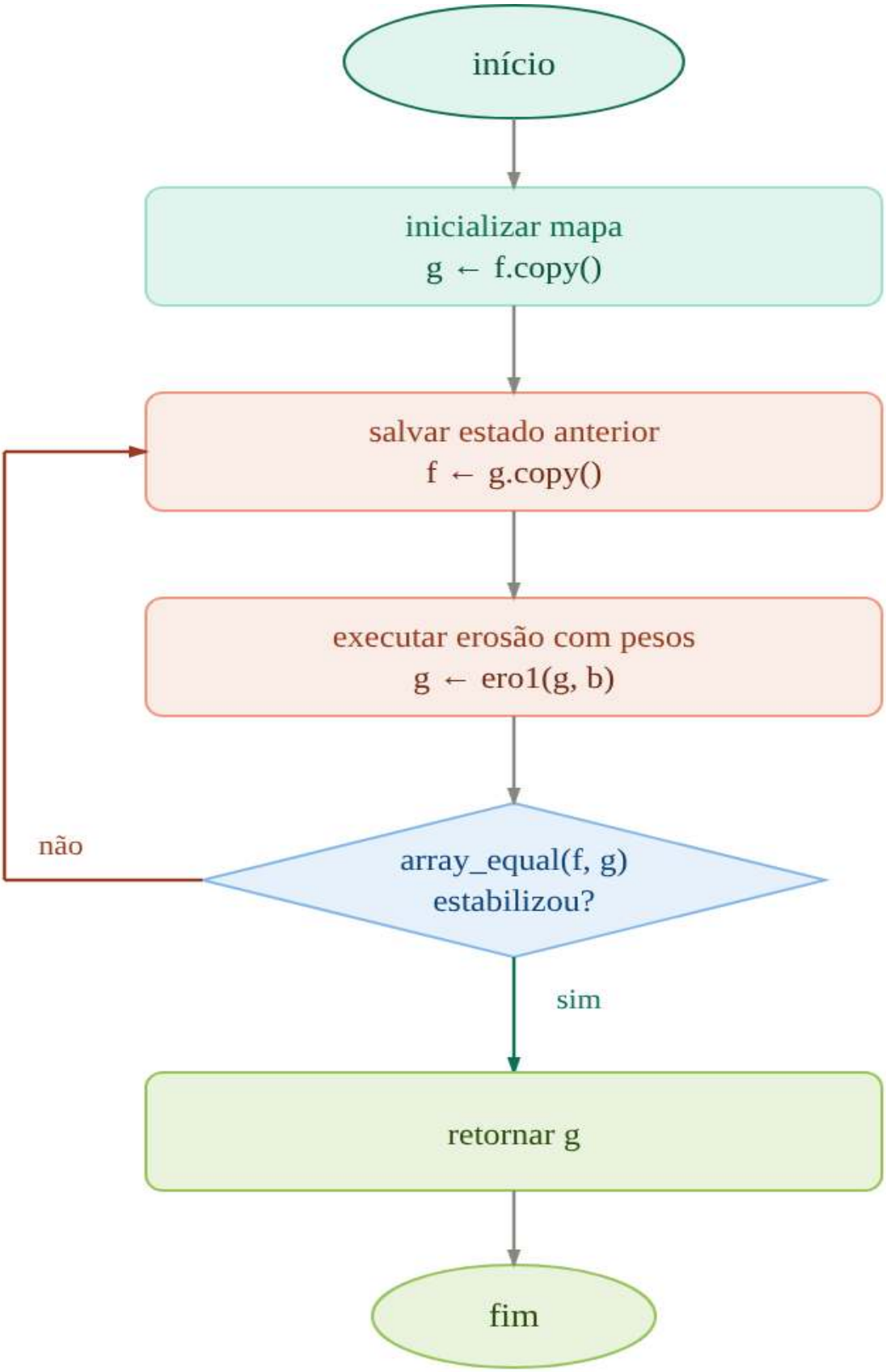
Esta interpretación se implementa en `mm.dist1()`, que acumula erosiones sucesivas utilizando la operación `mm.ero1()`. Por su parte, `mm.dist()` delega el cálculo de la distancia Euclidiana al operador optimizado de OpenCV `mm.dist(f)`, donde `f` es la imagen binaria de entrada, la distancia L2 especifica la métrica Euclidiana (L_2) y `5` indica el uso de una máscara 5×5 para aproximar la distancia con elevada precisión.

La función `dist1` produce una transformada de distancia discreta cuya métrica está determinada por la geometría y los pesos de la función estructurante utilizada. Por ejemplo, utilizando una función estructurante en cruz con costo unitario para los cuatro vecinos ortogonales, se obtiene la distancia de ***Manhattan*** (L_1). Otras elecciones de vecindad y pesos inducen métricas diferentes. En cambio, `mm.dist()` calcula una aproximación eficiente de la distancia Euclidiana (L_2).

Por exigir sucesivas erosiones sobre toda la imagen, el enfoque `dist1` posee un costo computacional significativamente mayor que `mm.dist()`, siendo empleado en este libro principalmente con fines didácticos y para evidenciar la relación entre morfología matemática y transformadas de distancia.

La Figura 4.19 presenta un simulador interactivo de la TD: es posible posicionar el cursor sobre diferentes píxeles del objeto y observar, en tiempo real, el valor de la distancia asociado a esa posición, es decir, la distancia hasta el píxel de fondo más cercano. La Figura 4.20 presenta un ejemplo práctico de esta ejecución en entorno Python.

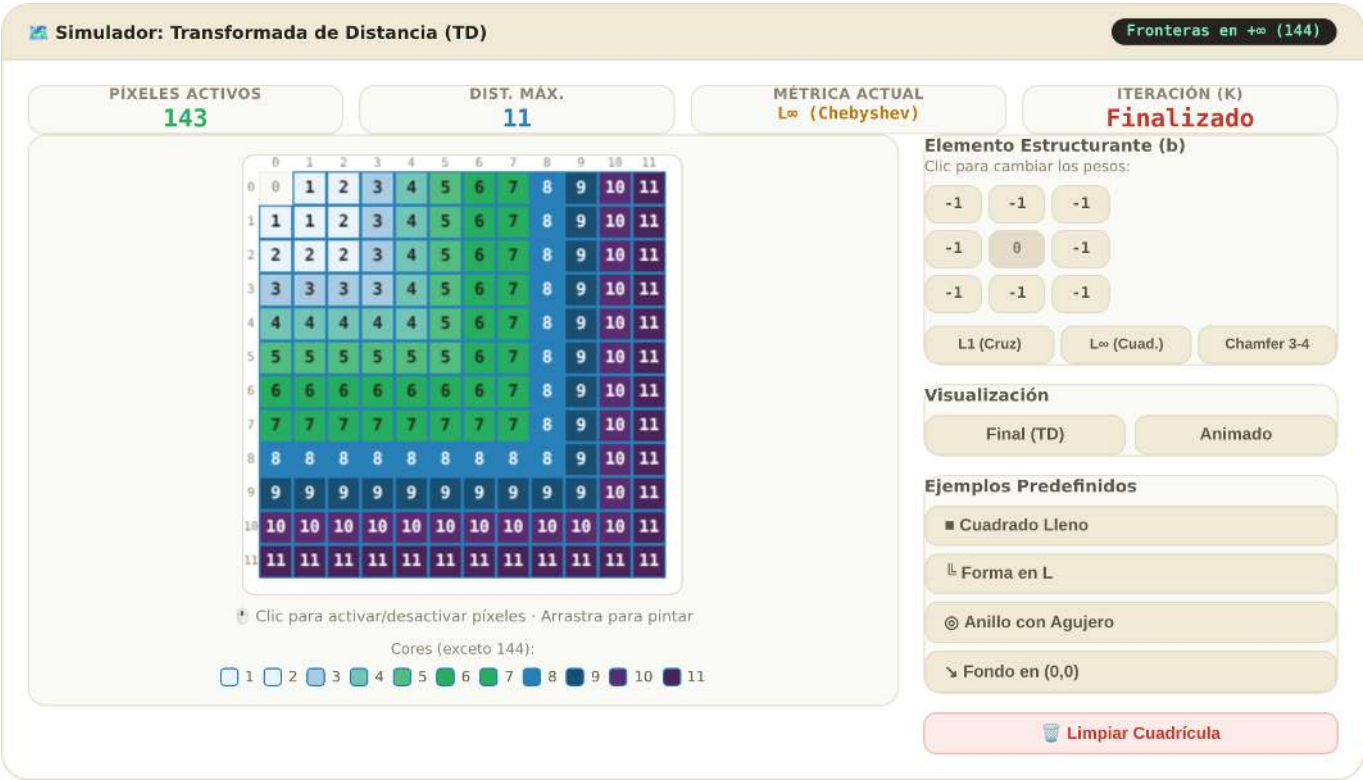
```
f = np.ones((10, 10), dtype=np.uint8) * 255
f[0, 0] = 0

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)
```



Ponto Fixo Numérico: A distância emerge da propagação matemática do valor zero.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-alg-distancia2.html

Figura 4.18: Algoritmo de la Transformada de Distancia.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-04-distancia.html

Figura 4.19: Simulador interactivo de la Transformada de Distancia (TD) iterativa mediante erosión en tonos de gris. Los píxeles fuera de la imagen asumen el valor máximo (144), propagando los costos a partir del fondo interno.

```
print(f"Máx. dist1 (erosiones) : {int(d_iter.max())} px")
print(f"Máx. dist  (L2)        : {int(d_l2.max())} px")

mm.show(
    [f, d_iter, d_l2],
    titles=["f original", "mm.dist1 (erosiones con cruz)", "mm.dist (L2)"],
    cols=3, figsize=(12, 4), axis=True
)
```

```
Máx. dist1 (erosiones) : 18 px
Máx. dist  (L2)        : 12 px
```

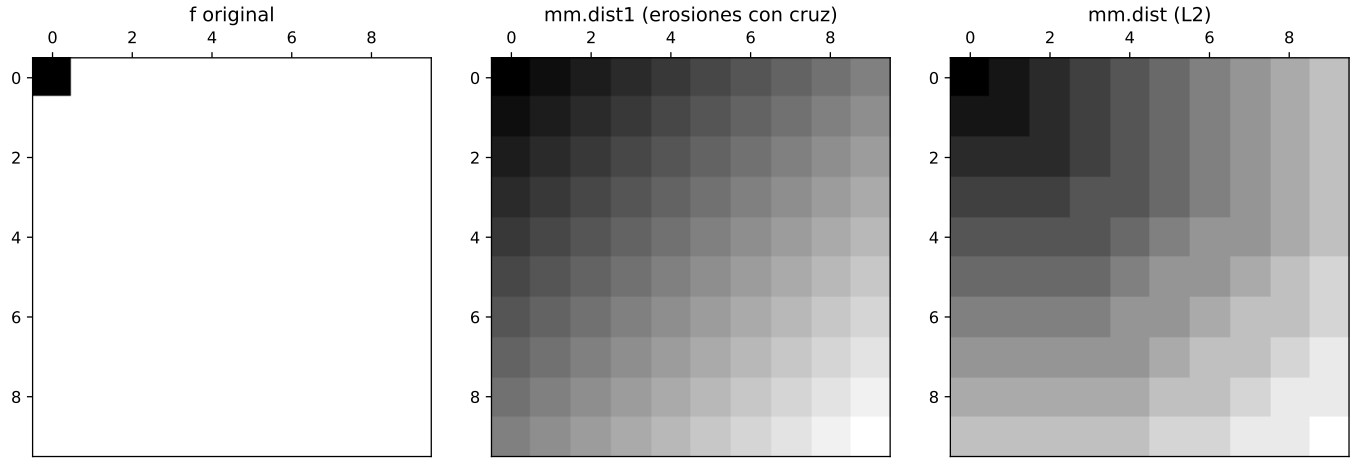


Figura 4.20: Transformada de Distancia en imagen binaria 10×10. Izquierda: original (*foreground* = 255). Centro: mm.dist1 iterativa (erosiones con cruz). Derecha: mm.dist (L2).

La anotación de los valores numéricos directamente sobre los píxeles permite verificar cómo `dist1` propaga las distancias según la métrica inducida por la función estructural utilizada. En el caso del elemento cruz con costo unitario, los valores obtenidos corresponden a la distancia de *Manhattan* (L_1). Aunque `dist1` y `mm.dist` producen valores numéricos distintos por adoptar métricas diferentes, ambas transformadas preservan la estructura topográfica de los objetos, haciendo que sus máximos ocurran en regiones centrales similares. Esta propiedad justifica el uso de `mm.dist` en aplicaciones prácticas, debido a su elevada eficiencia computacional.

4.4.3 Transformada de Distancia Euclidiana en cuatro pasos

El simulador de la Figura 4.21 implementa el algoritmo de la TDE de Lotufo (2001) en dos etapas. En la primera etapa, la función `edt1` realiza una transformación unidimensional vertical de forma secuencial (*in-place*), recorriendo cada columna en *raster* (↓) y *anti-raster* (↑) para calcular las distancias en la dirección vertical (dos pasos: Sur y Norte). En la segunda etapa, la función `edt2` utiliza ese resultado como entrada y realiza una propagación horizontal por colas: para cada fila de la matriz, dos colas de prioridad, `Eq` y `Wq`, se inicializan recorriendo los índices de columna en sentidos opuestos (`Eq` de `W-1` hasta `1`, `Wq` de `2` hasta `W`), de modo que cada píxel actualizado encola inmediatamente a sus vecinos para reprocesamiento dentro de la misma ronda (dos pasos más: Este y Oeste). Este mecanismo de cola permite aplicar erosiones sucesivas con pesos impares crecientes (`b = 1, 3, 5, ...`, incrementado en cada iteración del bucle externo) sin que la propagación quede atrapada en valores desactualizados, ya que cada ronda resuelve por completo la cadena de dependencias horizontales antes del siguiente incremento de `b`. La convergencia de esta propagación produce la Transformada de Distancia Euclidiana en toda la matriz, combinando la información vertical obtenida en `edt1` con la propagación horizontal en cola realizada en `edt2`.

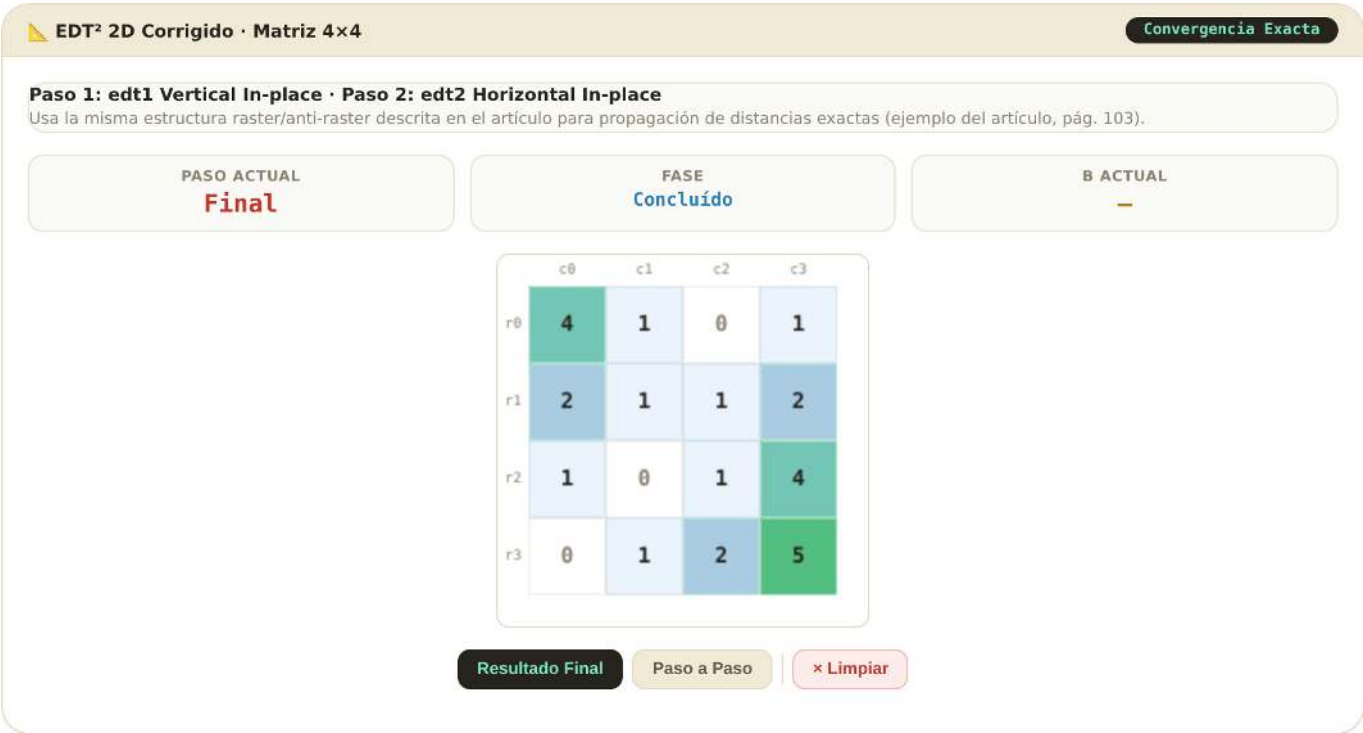
4.4.3.1 Transformada de Distancia Geodésica

La transformada de distancia geodésica asocia a cada píxel la menor distancia hasta un marcador, bajo la restricción impuesta por una máscara. De esta forma, la propagación ocurre exclusivamente por los píxeles permitidos, preservando la conectividad del dominio.

La Figura 4.22 ilustra este proceso en un laberinto: (a) la máscara `g`; (b) la distancia geodésica `D1` calculada a partir de la entrada; (c) la distancia `D2` calculada a partir de la salida; y (d) el camino mínimo obtenido a partir de estas dos transformadas.

El camino óptimo se determina mediante la suma de las distancias (`D1 + D2`). Los píxeles pertenecientes a la trayectoria mínima son aquellos para los cuales esta suma asume su menor valor, definiendo una conexión entre entrada y salida con longitud geodésica mínima.

Este principio permite resolver laberintos sin la necesidad de explorar explícitamente todas las posibilidades de recorrido. La solución emerge directamente de la propagación de distancias en un dominio restringido.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-04-tde.html

Figura 4.21: Simulador interactivo 2D (4x4) con sincronización estricta de colas de propagación horizontal (b) para obtener la convergencia exacta descrita en el artículo.

Este enfoque es particularmente relevante en laberintos de elevada complejidad, como los construidos a partir de estructuras cuasicristalinas y ciclos hamiltonianos descritos por Singh (2024). En Zampirolli (2025), este mismo formalismo se emplea para resolver un laberinto complejo; a continuación, el método se ilustra en una versión simplificada del problema.

```
import numpy as np

# 1 = pasillo, 0 = pared
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)

# marcador de la entrada
entrada = np.zeros_like(g, dtype=np.uint8)
entrada[0,1] = 1

# marcador de la salida
saida = np.zeros_like(g, dtype=np.uint8)
saida[9,8] = 1

# distancias geodésicas
D1 = mm.gdist(g, entrada)
D2 = mm.gdist(g, saida)
```

```
# suma de las distancias
S = D1 + D2

# menor valor válido de la suma
dmin = np.min(S[S > 0])

# píxeles pertenecientes a un camino óptimo
caminho = (S == dmin)

print("Distancia geodésica mínima:", dmin)

mm.show(
    [g, D1, D2, caminho],
    titles=[
        "Laberinto",
        "Distancia de la Entrada",
        "Distancia de la Salida",
        f"Menor Camino\n(d={dmin})"
    ],
    cols=4,
    figsize=(14,4),
    axis=True
)
```

```
Distancia geodésica mínima: 15
```

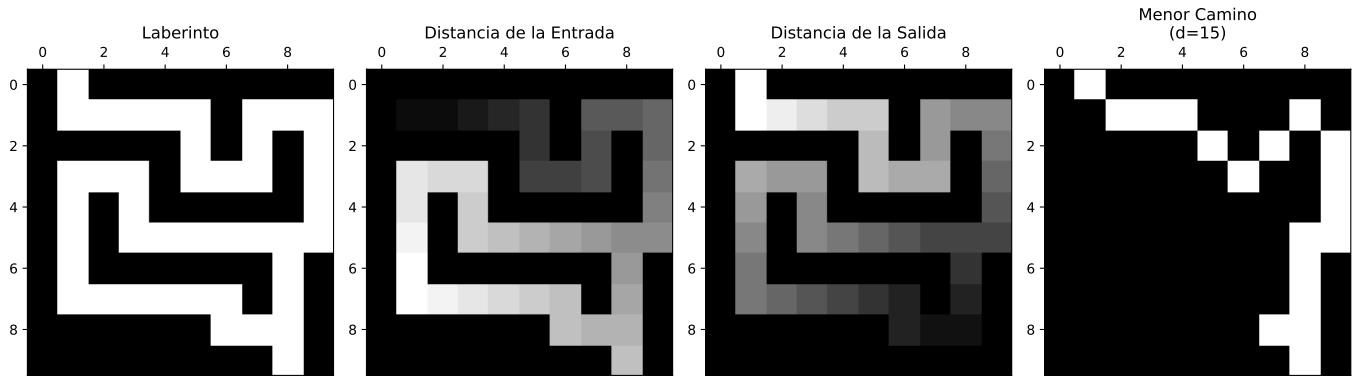


Figura 4.22: Camino geodésico más corto en un laberinto. Las distancias geodésicas se calculan desde la entrada y la salida usando mm.gdist. Los píxeles cuya suma de las dos distancias es igual a la distancia mínima entre los marcadores pertenecen a un camino óptimo.

4.4.4 Segmentación por *Watershed*

El algoritmo ***Watershed*** interpreta una imagen en niveles de gris como una superficie topográfica, en la cual valores elevados corresponden a montañas y valores bajos corresponden a valles o cuencas de drenaje (*catchment basins*). En el contexto de la segmentación basada en marcadores, los máximos de la Transformada de Distancia se utilizan frecuentemente para identificar regiones internas de los objetos, proporcionando semillas confiables para el proceso de inundación.

La segmentación se realiza entonces mediante una simulación conceptual de inundación progresiva a partir de estos marcadores. A medida que las cuencas asociadas a diferentes semillas se expanden, las regiones vecinas eventualmente entran en contacto. En ese instante se construyen barreras virtuales, denominadas *watershed lines*, que pasan a delimitar los objetos de la escena. Este mecanismo permite separar objetos adyacentes o parcialmente superpuestos, incluso cuando forman una única componente conexa tras la umbralización.

La implementación didáctica presentada en este capítulo explora inicialmente el concepto de crecimiento de

regiones (*region growing*) confinado por una máscara binaria, según se detalla en el algoritmo interactivo de la Figura 4.23.

> **Versión didáctica versus implementación clásica**
>
> La función `mm.watershed0` no implementa el algoritmo *watershed* clásico. Su objetivo es ilustrar, de forma simplificada, la propagación de marcadores por crecimiento de regiones (*region growing*), permitiendo visualizar cómo diferentes semillas compiten por la ocupación del espacio disponible. El crecimiento está delimitado por una máscara binaria de soporte y monitoreado por un control de estancamiento, generando un resultado similar a una partición de Voronoi restringida a la geometría de los objetos de entrada.
> En cambio, la función `mm.watershed` utiliza la implementación optimizada de OpenCV (`mm.watershed`), que realiza la inundación sobre una superficie topográfica definida por la imagen de entrada. En este caso, la propagación de los marcadores está influenciada por los valores de los píxeles, haciendo que las líneas de separación se formen naturalmente sobre las crestas del relieve.

A Figura 4.24 presenta un simulador iterativo que ilustra la propagación de los marcadores por la región de interés. Cada marcador actúa como una fuente de inundación que expande su área de influencia hasta encontrar regiones provenientes de otras semillas. En el algoritmo de OpenCV, los píxeles pertenecientes a las líneas divisorias se identifican por el valor `-1`, representando las fronteras entre cuencas adyacentes.

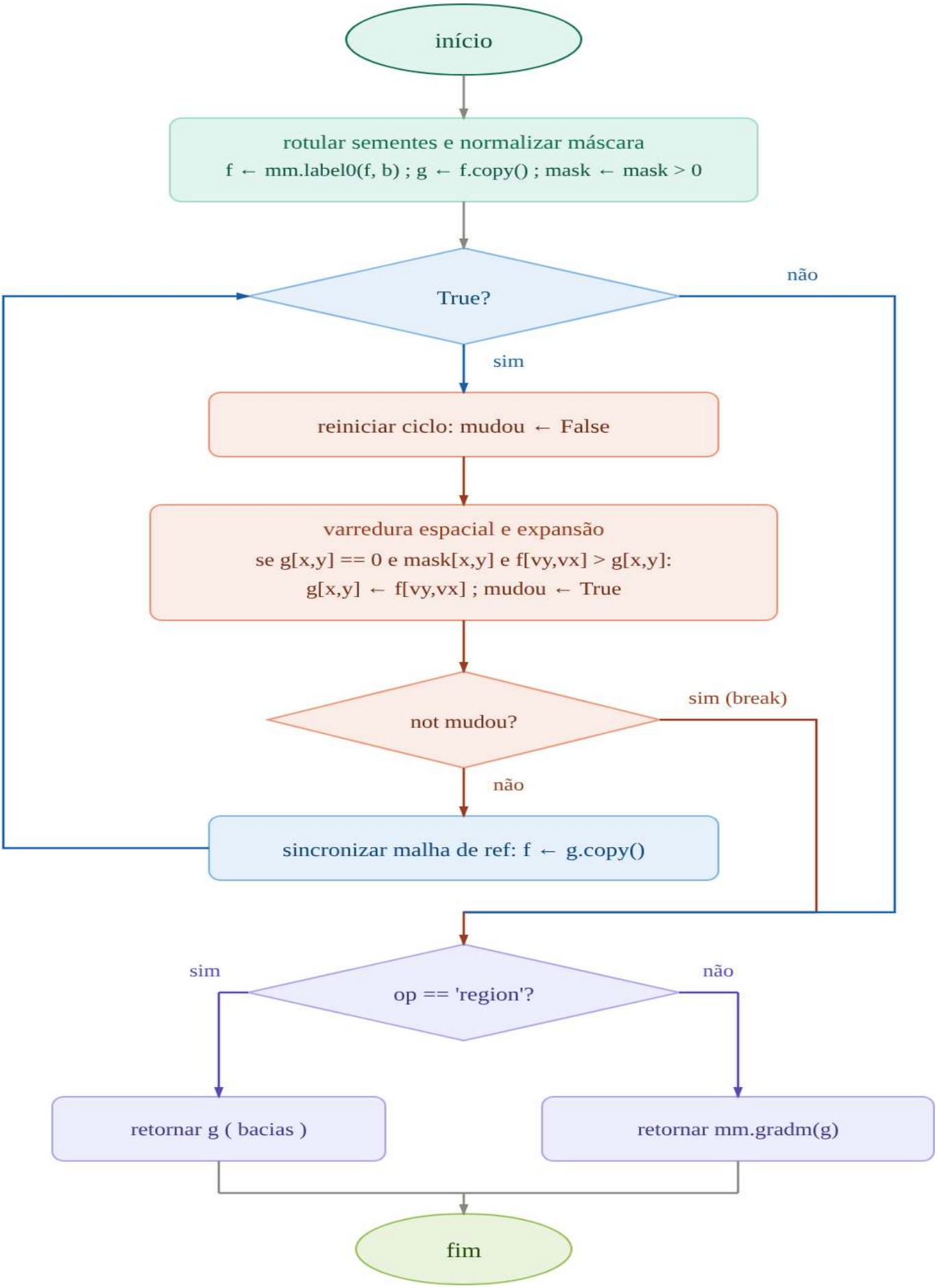
Pipeline Morfológico del *Watershed*

El *watershed* basado en marcadores normalmente integra un flujo más amplio de segmentación. En imágenes reales, etapas de preprocesamiento son frecuentemente necesarias para mejorar el contraste, reducir ruidos y generar marcadores confiables. Este flujo completo está resumido en la Tabla 4.6.

Tabla 4.6: *Pipeline* completo del *watershed* basado en marcadores para imágenes reales.

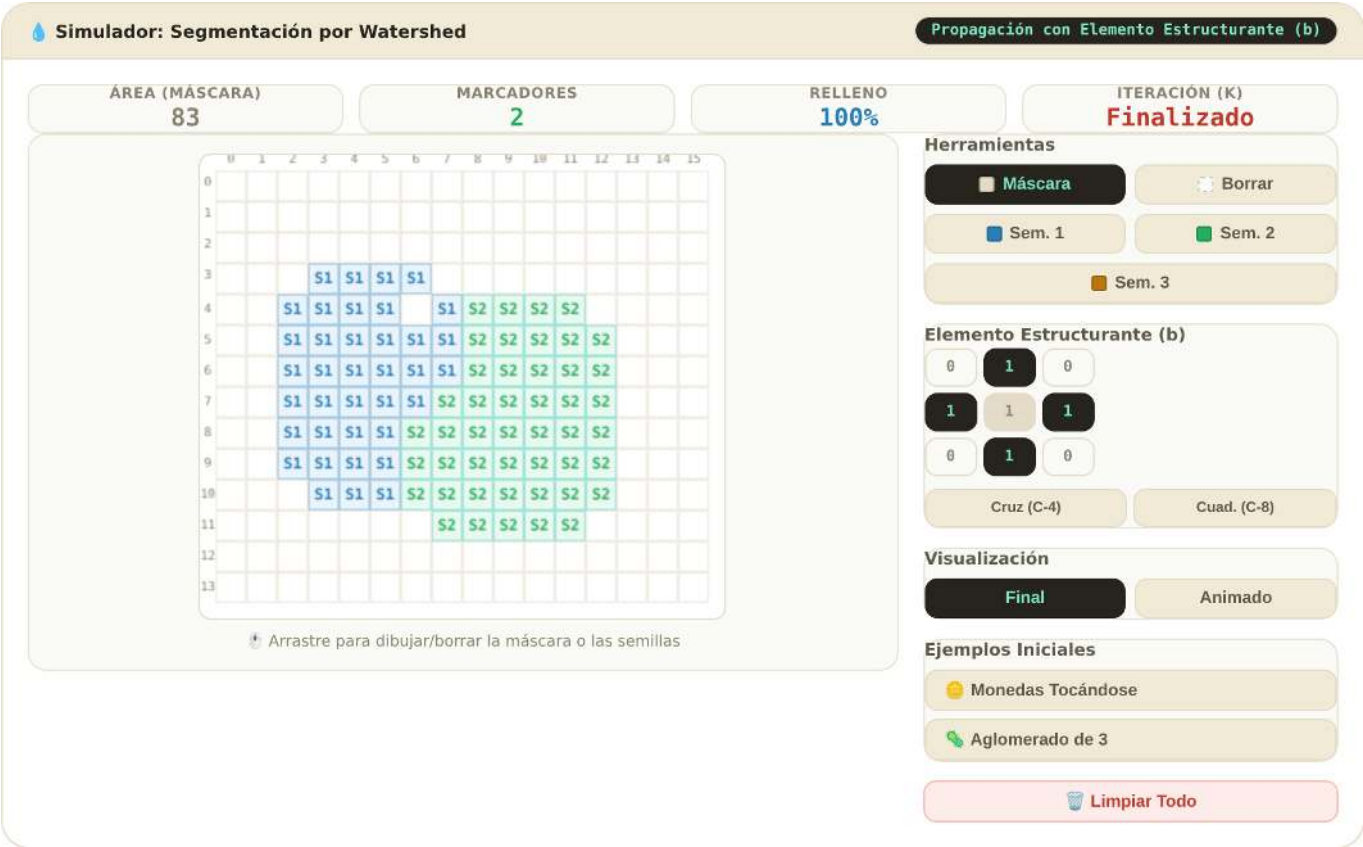
Etapa	Operación	Finalidad
1	CLAHE + Suavizado	Realce de contraste y reducción de ruido
2	Umbralización	Separación inicial entre objeto y fondo
3	Apertura/Cierre	Eliminación de ruidos y pequeñas imperfecciones
4	Dilatación de la Máscara	Identificación del Fondo Seguro (*Sure Background*)
5	Transformada de Distancia + Umbral	Identificación del Objeto Seguro (*Sure Foreground*)
6	Región Incierta	Diferencia entre Fondo Seguro y Objeto Seguro
7	`mm.watershed`	Propagación de los marcadores por la región incierta

Para enfatizar exclusivamente los conceptos de Transformada de Distancia, marcadores e inundación topográfica, el ejemplo de la Figura 4.25 utiliza una imagen binaria sintética y adopta un flujo simplificado, resumido en la Tabla 4.7.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-alg-watershed2.html

Figura 4.23: Algoritmo Didáctico de *Watershed* por Crecimiento de Regiones Limitado por Máscara.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-04-watershed.html

Figura 4.24: Simulador interactivo del Algoritmo *Watershed* por propagación morfológica. Dibuja la máscara, coloca los marcadores y ajusta el Elemento Estructurante para observar la inundación. Cuando las cuencas se encuentran simultáneamente, el empate se resuelve asumiendo una de las regiones de manera aleatoria.

Tabla 4.7: *Pipeline* simplificado utilizado en el ejemplo didáctico de la Figura 4.25.

Etapa	Operación	Finalidad
1	Transformada de Distancia	Construcción de la superficie topográfica
2	Umbral de la TD	Extracción de los marcadores (*Sure Foreground*)
3	Dilatación	Determinación del Fondo Seguro (*Sure Background*)
4	Región Incierta	Diferencia entre fondo y marcadores
5	`mm.watershed`	Propagación de los marcadores y generación de las fronteras

```
# 1. Imagen sintética y Transformada de Distancia
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. Marcadores (picos de la distancia)
m = (dist > 0.8 * dist.max()).astype(np.uint8) * 255

# 3. Ejecución del Watershed0 condicional (m=marcadores primero, 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. Exhibición de los resultados
mm.show([f_sint, dist, m, w_reg, w_line], cols=5, figsize=(16, 4),
        titles=["Original", "Distancia", "Marcadores", "Regiones", "Líneas"])
```

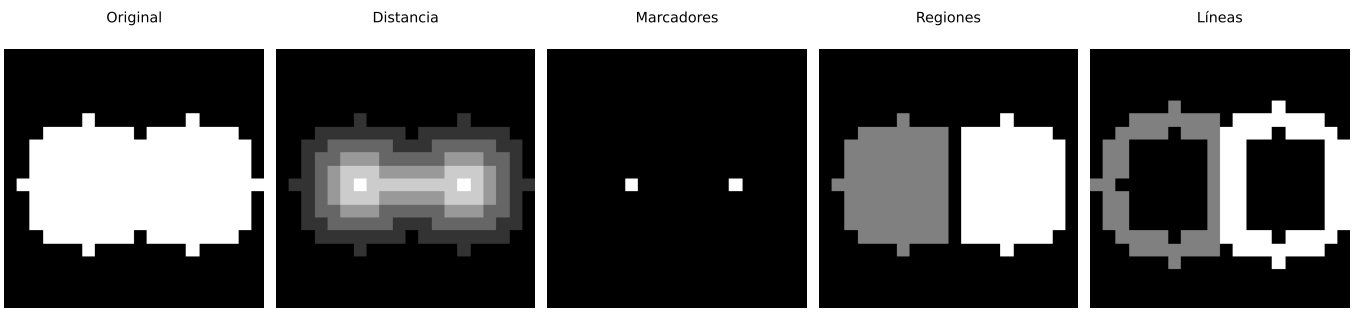


Figura 4.25: *Pipeline watershed* delimitado por máscara en imagen binaria 20×20.

Aplicación en monedas superpuestas:

```
img_base = img_coins_gray

# 1. Simular monedas superpuestas/conectadas (usando la máscara binaria base)
img_sobrepostas = mm.dil(img_final, mm.sebox(40))
```

```python
# 2. Apertura morfológica para limpiar ruidos
opening = mm.open(img_sobrepostas, mm.sebox(2))

# 3. Transformada de Distancia
dist = mm.dist(opening)
dist_vis=(255*(dist/dist.max())).astype(np.uint8) if dist.max() > 0 else dist.astype(np.uint8)

# 4. Picos seguros (Marcadores de las monedas)
picos = (dist > 0.5 * dist.max()).astype(np.uint8) * 255

# 5. Ejecución del Watershed (Ajustado para usar la nueva firma)
# Pasamos 'opening' directo en mask, pues delimita el alcance de expansión de las monedas
ws_region = mm.watershed(picos, mask=opening, op='region')
ws_line   = mm.watershed(picos, mask=opening, op='line')

# 6. Conteo de objetos (Ignora fondo 0)
labels = np.unique(ws_region)
labels = labels[labels > 0]
print(f"Objetos detectados: {len(labels)}")

# 7. Anotación Final
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)

# Dibuja las líneas de separación en rojo
mask_ann = mm.dil(ws_line, np.ones((11, 11), np.uint8))
img_annotated[mask_ann > 0 ] = [255, 0, 0]

# Visualización
mm.show(
    [img_base, img_sobrepostas, dist_vis, picos, ws_region, img_annotated],
    titles=["Original", "Superpuestas", "Distancia", "Marcadores", "Watershed", "Anotado"],
    cols=3, rows=2, figsize=(12, 8)
)
```

```
Objetos detectados: 12
```

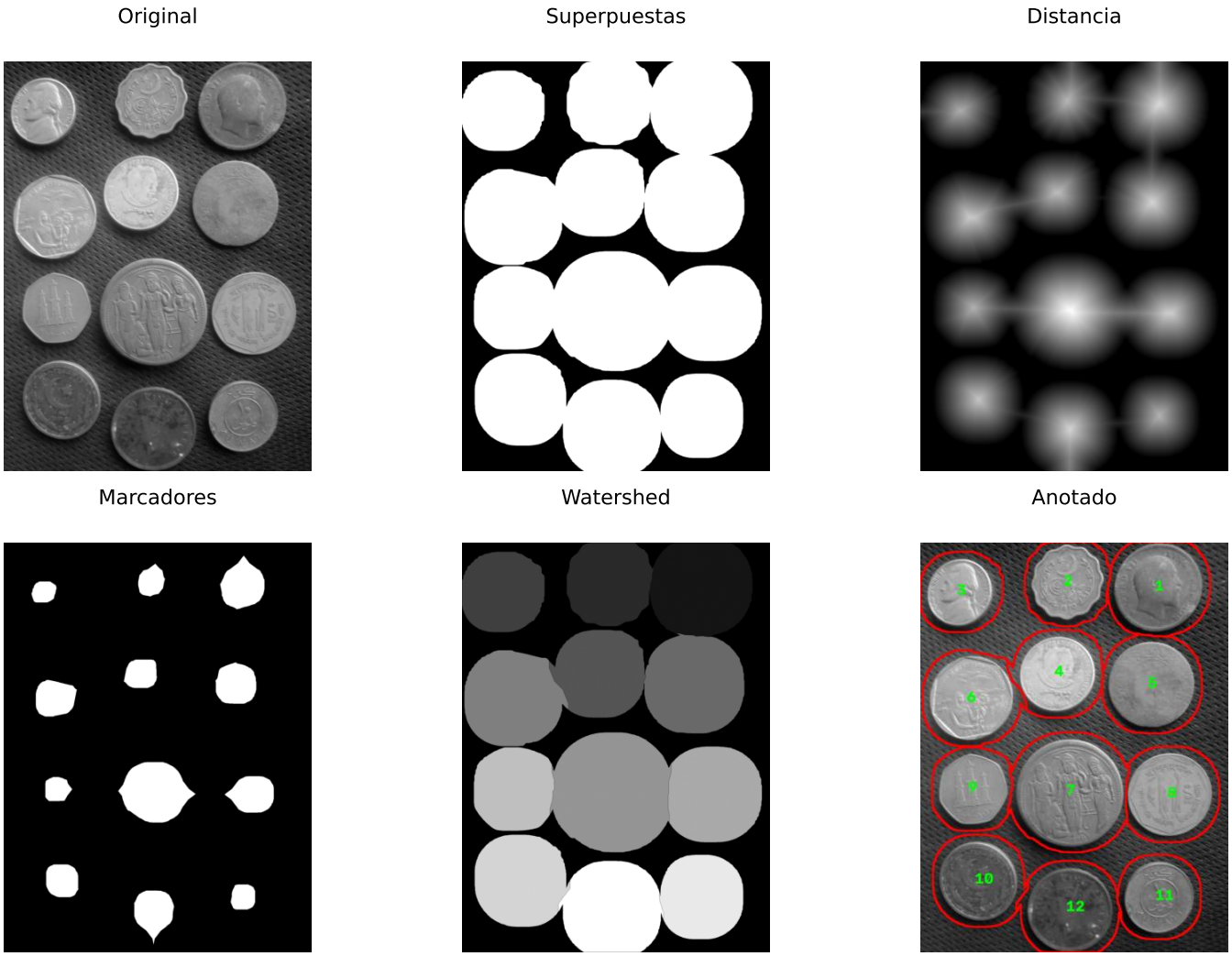


Figura 4.26: *Pipeline Watershed* para separación de monedas superpuestas: desde la máscara binaria dilatada hasta los contornos finales anotados sobre la imagen original.

4.5 Extracción de Componentes y Descriptores de Forma

Tras la segmentación y el refinamiento morfológico, el siguiente paso consiste en identificar individualmente cada objeto presente en la imagen y extraer sus propiedades geométricas. Esta etapa es fundamental para tareas de medición, clasificación y reconocimiento de patrones.

Un **componente conexo** es un conjunto maximal de píxeles pertenecientes al objeto que permanecen mutuamente conectados según una relación de conectividad previamente definida (conectividad 4 u 8). Tras la rotulación, cada componente recibe un identificador único, permitiendo que sus características sean analizadas individualmente.

OpenCV ofrece dos enfoques complementarios para este análisis, resumidos en la Tabla 4.8.

Tabla 4.8: Comparación entre los enfoques basados en componentes conexos y en contornos.

	`connectedComponentsWithStats`	`findContours`
Devuelve	etiqueta por píxel y estadísticas por componente	secuencia de puntos que describe el borde
Descriptores directos	área, *bounding box* y centroide	perímetro, forma y jerarquía
Objetos en contacto	tiende a fusionar regiones conectadas	tiende a producir un único contorno externo

	connectedComponentsWithStats	findContours
Uso típico	conteo, filtrado y rotulación	análisis geométrico y descriptores de forma

4.5.1 Rotulado y Estadísticas de Componentes

La función `mm.label0` realiza simultáneamente el rotulado de los componentes conexos y la extracción de descriptores básicos de cada región. El operador devuelve:

- una imagen de etiquetas (`labels`);
- estadísticas geométricas (`stats`);
- coordenadas de los centroides (`centroids`).

Las estadísticas incluyen área, ancho, alto y posición de la *bounding box* mínima alineada con los ejes de la imagen.

La Figura 4.27 ilustra la aplicación de este operador después del *pipeline* de segmentación de las monedas. Cada componente conexo recibe un color distinto y su área se anota directamente sobre la imagen.

```
# 1. Etiquetado y estadísticas
n, labels, stats, centroids = cv2.connectedComponentsWithStats(
    img_final, connectivity=8
)

# 2. Coloración de los componentes - paleta FIJA (generada una vez con
# np.random.seed(4)) para que la pista C++ reproduzca color a color sin
# depender del generador de numpy. Si n excede len(PALETA), cicla.
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]                          # fondo negro
img_colored   = colors[labels]
img_annotated = img_colored.copy()

# 3. Tabla de descriptores
print(f"Componentes detectados (excluyendo fondo): {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", "Segmentación Final", "Componentes Conexos", "Áreas Anotadas"],
    cols=4, figsize=(18, 6)
)
```

```
Componentes detectados (excluyendo fondo): 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
```

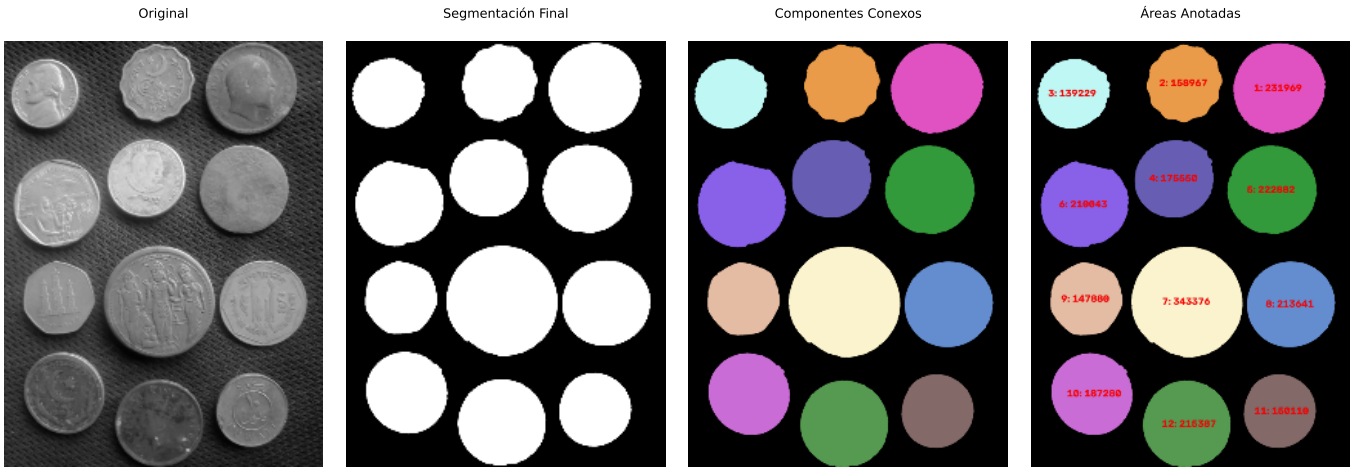


Figura 4.27: Componentes conexos extraídos tras el *pipeline* CLAHE → Otsu → limpieza morfológica. Cada objeto se colorea con color distinto y se anota con su área en píxeles.

4.5.2 Descriptores de Forma

Mientras que `connectedComponentsWithStats` opera sobre regiones, la extracción de contornos actúa directamente sobre sus fronteras. El operador devuelve, para cada objeto, una secuencia ordenada de puntos que describe su contorno.

A partir de esta representación es posible calcular descriptores geométricos que no se proporcionan directamente mediante la etiquetación de componentes:

- **Área** (área del contorno)
- **Perímetro** (perímetro);
- **Circularidad** ($C = \frac{4\pi A}{P^2}$, donde A es el área y P el perímetro);
- **Aproximación poligonal** (aproximación poligonal);
- **Casco convexo** (*convex hull*);
- **Jerarquía de contornos**, que permite representar relaciones padre-hijo entre regiones y sus agujeros internos.

La circularidad alcanza un valor máximo igual a 1 para un círculo perfecto y disminuye a medida que el objeto se vuelve más alargado o presenta irregularidades en su borde.

La Figura 4.28 presenta los contornos extraídos de las monedas segmentadas, junto con los valores de circularidad calculados para cada objeto.

```python
contornos, hierarquia = cv2.findContours(
    img_final, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
)

img_contornos  = cv2.cvtColor(img_coins_gray, cv2.COLOR_GRAY2BGR)
img_circulares = img_contornos.copy()

print(f"Contornos detectados: {len(contornos)}")
print(f"\n{'ID':>4} {'Área':>8} {'Perímetro':>10} {'Circularidade':>14}")
print("-" * 42)

for i, cnt in enumerate(contornos, start=1):
    area  = cv2.contourArea(cnt)
    perim = cv2.arcLength(cnt, closed=True)
    circ  = (4 * np.pi * area / perim**2) if perim > 0 else 0
    M     = cv2.moments(cnt)
    cx    = int(M["m10"] / M["m00"]) if M["m00"] > 0 else 0
    cy    = int(M["m01"] / M["m00"]) if M["m00"] > 0 else 0

    print(f"{i:>4} {area:>8.0f} {perim:>10.1f} {circ:>14.3f}")

    cv2.drawContours(img_contornos,  [cnt], -1, (0, 255, 0), 3)
    cv2.drawContours(img_circulares, [cnt], -1, (0, 255, 0), 3)
    for cor, esp in [((0,0,0), 8), ((255,0,0), 3)]:
        cv2.putText(img_circulares, f"{circ:.2f}",
                    (cx - 80, cy + 15),
                    cv2.FONT_HERSHEY_SIMPLEX, 3.6, cor, esp, cv2.LINE_AA)

mm.show(
    [img_final, img_contornos, img_circulares],
    titles=["Segmentación Final", "Contornos", "Circularidad"],
    cols=3, figsize=(18, 6)
)
```

```
Contornos detectados: 12

  ID     Área  Perímetro  Circularidade
------------------------------------------
   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
```

Figura 4.28: Contornos extraídos con *findContours*. Cada moneda se anota con su circularidad — valores cercanos a 1 confirman forma circular.

4.5.3 Conexión con la Detección de Objetos Moderna

Los descriptores extraídos en las secciones anteriores — especialmente *bounding boxes*, centroides, áreas y medidas de forma — establecen un puente natural entre la segmentación morfológica clásica y los sistemas modernos de detección de objetos. Aunque las técnicas estudiadas en este capítulo utilizan operaciones sobre píxeles y regiones segmentadas, muchas de las representaciones producidas son directamente compatibles con los formatos empleados en modelos contemporáneos de visión por computadora.

Los detectores basados en aprendizaje profundo, como la familia YOLO (*You Only Look Once*) (REDMON, 2016), operan directamente sobre imágenes en color y producen, para cada objeto detectado, una *bounding box* descrita por el centro (cx, cy) y las dimensiones (w, h), además de una clase y una puntuación de confianza. Esta representación comparte la misma estructura geométrica básica obtenida por `connectedComponentsWithStats`, aunque es producida por un modelo aprendido y no mediante segmentación explícita.

La Figura 4.29 ilustra cómo las *bounding boxes* obtenidas por morfología pueden exportarse en el formato YOLO para componer conjuntos de datos utilizados en el entrenamiento o la evaluación de detectores.

```
# Recalcula etiquetas/estadísticas a partir de la segmentación final
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 = moneda (única categoría en este ejemplo)

print(f"{'cls':>4} {'cx_n':>8} {'cy_n':>8} {'w_n':>8} {'h_n':>8}  ← formato YOLO")
print("-" * 54)

yolo_linhas = []
for i in range(1, n):
    x0 = stats[i, cv2.CC_STAT_LEFT]
    y0 = stats[i, cv2.CC_STAT_TOP]
    w  = stats[i, cv2.CC_STAT_WIDTH]
    h  = stats[i, cv2.CC_STAT_HEIGHT]

    cx_n = (x0 + w / 2) / W_img
```

```
    cy_n = (y0 + h / 2) / H_img
    w_n  = w / W_img
    h_n  = h / H_img

    yolo_linhas.append(f"{CLASSE} {cx_n:.4f} {cy_n:.4f} {w_n:.4f} {h_n:.4f}")
    print(f"{CLASSE:>4} {cx_n:>8.4f} {cy_n:>8.4f} {w_n:>8.4f} {h_n:>8.4f}")

    cv2.rectangle(img_bbox, (x0, y0), (x0 + w, y0 + h), (0, 255, 0), 4)
    cv2.putText(img_bbox, f"moeda", (x0 + 8, y0 + 60),
                cv2.FONT_HERSHEY_SIMPLEX, 3.8, (255, 0, 0), 5, cv2.LINE_AA)

# Exportar archivo de anotación en el formato YOLO
with open("moedas.txt", "w") as f:
    f.write("\n".join(yolo_linhas))
print("\nAnotación guardada en monedas.txt")

mm.show(
    [img_coins_gray, img_final, img_bbox],
    titles=["Original", "Segmentación Final", "Bounding Boxes (formato YOLO)"],
    cols=3, figsize=(18, 6)
)
```

```
 cls     cx_n     cy_n      w_n      h_n  ← formato YOLO
-------------------------------------------------------
   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

Anotación guardada en monedas.txt
```

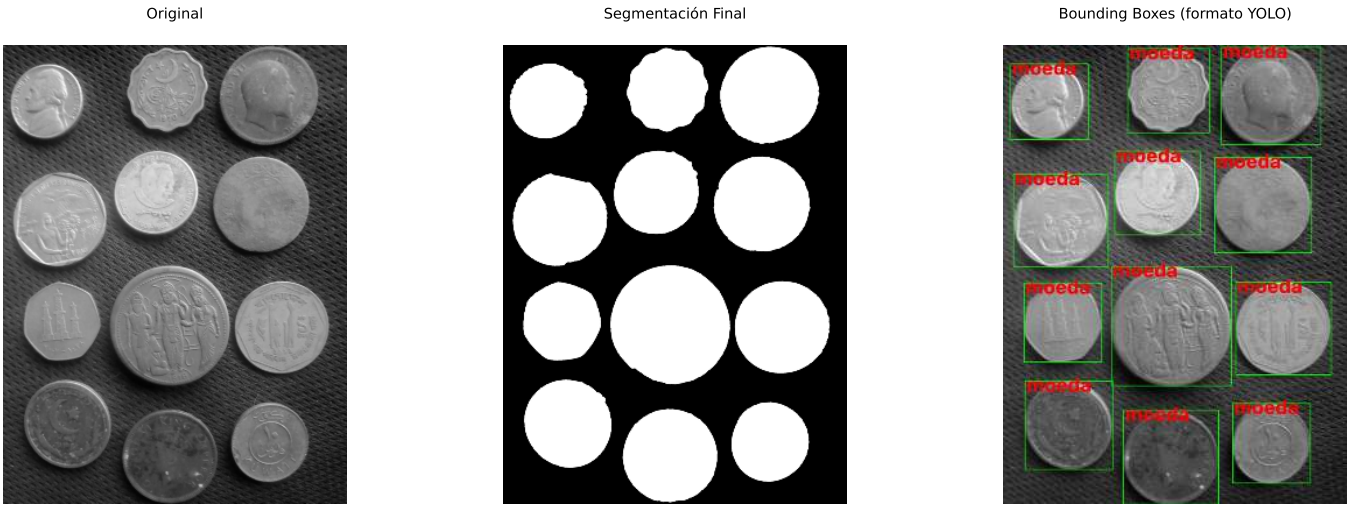


Figura 4.29: *Bounding boxes* derivadas de los componentes conectados superpuestas a la imagen original. Las anotaciones se exportan en el formato YOLO (*clase cx cy w h*), con coordenadas normalizadas al intervalo [0,1].

El formato YOLO almacena cada objeto en una línea que contiene cinco campos:

```
clase cx cy w h
```

donde (cx, cy) representa el centro de la *bounding box* y (w, h) sus dimensiones. Todos los valores geométricos se normalizan al intervalo $[0, 1]$ con respecto al ancho y al alto de la imagen. La **clase** es un identificador entero asociado a una categoría definida por el conjunto de datos (por ejemplo, `0 → moneda`). Cuando hay múltiples categorías —como moneda de oro (`0`), moneda de plata (`1`) y disco plástico (`2`)— basta con asignar el identificador correspondiente a cada objeto antes de la exportación, manteniendo exactamente el mismo formato de anotación. En el ejemplo anterior, esta información se almacenó en el archivo `monedas.txt`.

El flujo presentado en este capítulo —segmentación → etiquetado → extracción de *bounding boxes*— corresponde conceptualmente a la etapa de **anotación** (*labeling*) empleada en la construcción de conjuntos de entrenamiento para detectores modernos. Herramientas especializadas, como Label Studio y Roboflow, automatizan este proceso en imágenes complejas, pero la lógica fundamental sigue siendo la misma: asociar a cada objeto una región de interés y una clase. En escenarios controlados, con fondo uniforme y objetos bien separados, las técnicas morfológicas pueden incluso generar anotaciones automáticamente o servir como punto de partida para el etiquetado manual, reduciendo significativamente el esfuerzo de construcción del conjunto de datos. En aplicaciones reales más complejas, sin embargo, la validación humana sigue siendo necesaria para garantizar la calidad de las anotaciones.

Evaluación: IoU (*Intersection over Union*)

Una forma sencilla de evaluar la calidad de una segmentación consiste en compararla con una máscara de referencia (*ground truth*). La métrica más utilizada para este propósito es la **IoU** (*Intersection over Union*):

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} \tag{4.16}$$

donde A representa la segmentación producida por el algoritmo y B la segmentación de referencia.
El valor de la IoU varía entre 0 y 1. Cuanto mayor sea el valor, mayor será la superposición entre las máscaras. Una IoU igual a 1 indica una correspondencia perfecta entre la segmentación obtenida y la referencia.
La misma métrica también se utiliza ampliamente en la detección de objetos, aplicándose a las *bounding boxes* previstas y anotadas. En esta área, valores de IoU superiores a 0,5 se adoptan frecuentemente como criterio mínimo para considerar una detección correcta.

Más Allá de la Morfología

Las técnicas estudiadas en este capítulo segmentan objetos explorando la conectividad espacial, las operaciones morfológicas y el relieve topográfico. Existen, sin embargo, enfoques alternativos basados en la agrupación de características, como el algoritmo *k-means*, los modelos de mezcla Gaussiana (GMM) y métodos más recientes basados en aprendizaje profundo. Estas técnicas se retomarán en la Parte II del libro, dedicada a la Visión Computacional.

4.6 Resumen

En este capítulo se presentaron las principales técnicas de segmentación y morfología matemática, concluyendo el estudio del PDI en el dominio espacial.

- **Preprocesamiento y umbralización:** La combinación entre ecualización adaptativa CLAHE y el método de Otsu mostró ser eficaz para inducir separación bimodal en el histograma y simplificar la binarización de imágenes con iluminación no uniforme.

- **Erosión y dilatación:** Operadores morfológicos fundamentales basados en la búsqueda de mínimos y máximos locales en una vecindad definida por el elemento estructurante B. Son operadores duales por el complemento y fueron implementados mediante las funciones `mm.ero` y `mm.dil`.
- **Apertura y cierre:** Composiciones de erosión y dilatación que permiten eliminar ruidos, suavizar contornos y rellenar pequeñas lagunas, preservando la estructura global de los objetos.
- **Reconstrucción morfológica:** Proceso geodésico iterativo que propaga un marcador dentro de los límites impuestos por una máscara, constituyendo la base de operadores como `mm.clohole` y `mm.edgeoff`.
- **Pipeline de limpieza binaria:** Flujo consolidado compuesto por CLAHE → Otsu → apertura → `mm.clohole` → apertura restringida → `mm.edgeoff`, produciendo máscaras adecuadas para análisis cuantitativo.
- **Morfología en tonos de gris:** Extensión algebraica basada en mínimos y máximos ponderados, viabilizando operadores como gradiente morfológico y filtros *top-hat* para realce de estructuras locales.
- **Transformada de Distancia y Watershed:** La Transformada de Distancia permitió generar marcadores automáticos para el algoritmo *watershed*, posibilitando la separación de objetos adyacentes o parcialmente superpuestos.
- **Componentes conexos y descriptores:** La rotulación de regiones (`mm.label0`) y la extracción de contornos permitieron calcular descriptores geométricos como área, centroide, perímetro, circularidad y *bounding boxes*.
- **Conexión con Visión Computacional Moderna:** Las *bounding boxes* extraídas por morfología fueron exportadas en el formato YOLO, evidenciando el vínculo entre técnicas clásicas de segmentación y sistemas modernos de detección de objetos.

El Capítulo 5 introducirá técnicas de procesamiento en el **dominio de la frecuencia**, abordando la **Transformada de Fourier**, filtrado espectral y los fundamentos de la **compresión de imágenes**, incluyendo DCT, JPEG y *wavelets*.

4.7 🤖 Uso de Gemini Notebook como Tutor Complementario

En esta edición, se incentiva el uso de **Gemini Notebook** como herramienta complementaria de aprendizaje. Basado en inteligencia artificial, el sistema utiliza exclusivamente los documentos proporcionados por el autor como fuente de conocimiento, produciendo respuestas alineadas con el contenido y el enfoque adoptado a lo largo de este capítulo.

🎓 Estudia con el Tutor Inteligente

🚀 ACCEDER A Gemini Notebook: CAPÍTULO 04

🌐 Idioma y Lenguaje de Programación

El proyecto de este capítulo en Gemini Notebook fue construido únicamente con el texto en **portugués** y los ejemplos de código en **Python**. Si estás estudiando con la edición en inglés o francés, o siguiendo la ruta en C++, las respuestas del tutor pueden no corresponder exactamente a la versión que estás leyendo.

⚠️ Aviso sobre Contenido Generado por IA

Aunque es una herramienta valiosa de apoyo al estudio, Gemini Notebook puede eventualmente producir respuestas incompletas, imprecisas o incorrectas. Se recomienda validar la información consultando el material del capítulo, libros, artículos científicos y otras fuentes académicas confiables. Siempre que sea posible, ejecuta y experimenta con los ejemplos prácticos presentados a lo largo del texto para consolidar la comprensión de los conceptos.

4.8 Lista de Ejercicios

1. **(10%)** Implemente manualmente el criterio de Otsu sin utilizar `mm.threshold`. Calcule la varianza entre clases $\sigma_B^2(T)$ para todos los umbrales $T \in [0, 255]$ usando `mm.hist`, identifique el umbral óptimo T^* y compare el resultado con el valor obtenido por OpenCV. Grafique σ_B^2 en función de T y resalte el punto de máximo.

2. **(15%)** Aplique umbralización adaptativa con bloques de tamaño 11, 31 y 51 a una imagen que contenga iluminación no uniforme. Compare los resultados con la umbralización global de Otsu y discuta las ventajas y limitaciones de cada enfoque.

3. **(15%)** Ejecute el *pipeline* de watershed de la imagen de monedas variando el umbral aplicado a la Transformada de Distancia (0.3, 0.5 y 0.7 veces el valor máximo). Explique cómo este parámetro influye en la generación de los marcadores, la separación de objetos adyacentes y la ocurrencia de sobre-segmentación.

4. **(15%)** Utilizando `mm.drawImg`, construya una demostración visual paso a paso de la erosión de una imagen binaria 7×7 con elemento estructurante cuadrado 3×3. Para cada posición analizada, indique si el elemento estructurante está completamente contenido en el objeto y justifique el valor asignado al píxel de salida.

5. **(15%)** Demuestre experimentalmente la dualidad entre erosión y dilatación verificando la identidad $(A \ominus B)^c = A^c \oplus \hat{B}$ utilizando `mm.ero`, `mm.dil` y `mm.bnot`. Calcule la diferencia píxel a píxel entre los dos lados de la ecuación y presente el resultado utilizando `mm.histImg` o una visualización equivalente.

6. **(15%)** Implemente manualmente el gradiente morfológico utilizando únicamente `mm.ero` y `mm.dil`, comparando el resultado con `mm.gradm(img, B)`. Evalúe el efecto de diferentes elementos estructurantes (cuadrado 3×3, disco 5×5 y línea 1×9) sobre la detección de bordes.

7. **(15%)** Construya un *pipeline* completo para el conteo y clasificación de monedas por tamaño (pequeña, mediana y grande) utilizando área y circularidad como descriptores. Genere una máscara de referencia (*ground truth*) manualmente y calcule la métrica IoU (*Intersection over Union*) para evaluar la calidad de la segmentación. Presente los resultados en una tabla y mediante visualizaciones producidas con `mm.show`.

Referencias del Capítulo

La fundamentación teórica de este capítulo se basa en las siguientes obras:

- Gonzalez (2018) para los conceptos de segmentación, umbralización de Otsu, Transformada de Distancia, *watershed*, morfología matemática y descriptores de forma.
- Matheron (1975) y Serra (1982) para la fundamentación teórica original, formulación algebraica y desarrollo de la Morfología Matemática.
- Szeliski (2022) para segmentación basada en regiones, etiquetado de componentes conexos, *watershed* basado en marcadores y evaluación de segmentación mediante la métrica IoU.
- Bradski (2008) para la utilización práctica de la biblioteca OpenCV, incluyendo funciones como `mm.dist`, `mm.watershed`, `mm.label0` y extracción de contornos.
- Redmon (2016) para la introducción a los detectores modernos de la familia YOLO y su relación con descriptores geométricos como *bounding boxes* extraídas por segmentación.
- Singh (2024) para la construcción de laberintos complejos basados en ciclos hamiltonianos sobre mosaicos cuasicristalinos, utilizados como ejemplo de aplicación de la Transformada de Distancia Geodésica y de algoritmos de búsqueda de caminos.
- Zampirolli (2025) para la implementación de los operadores morfológicos, transformadas geodésicas y resolución de laberintos por propagación de distancias en dominios restringidos.

Executar Colab | Abrir GitHub

4.9 EP4: Adición de Ruido y Filtros

Objetivo: Este ejercicio práctico (EP) tiene como objetivo ejercitar la aplicación de diferentes tipos de ruido a imágenes y el uso de filtros espaciales (media, mediana y gaussiano) para la restauración de imágenes degradadas. El estudiante deberá explorar los efectos de estos filtros sobre imágenes con ruido, comparando cualitativamente los resultados y entendiendo las características de cada técnica.

4.10 Cómo usar este notebook

Este notebook contiene celdas de código y celdas de texto. Las celdas de código deben ejecutarse en orden, de arriba hacia abajo, para que las variables y funciones definidas estén disponibles para las celdas posteriores. Para ejecutar una celda de código, selecciónela y presione `Shift+Enter`, o haga clic en el botón de ejecución en la barra de herramientas de Colab.

Las secciones marcadas como **EJERCICIOS** requieren que el estudiante complete el código (escriba una respuesta basada en los estudios previos). No se olvide de guardar una copia en su Google Drive (Archivo → Guardar una copia en Drive) antes de comenzar a editar. Para obtener una nota, será necesario completar todos los ejercicios propuestos y seguir las instrucciones dadas por el profesor.

4.11 Bibliotecas necesarias

A continuación se importan las bibliotecas necesarias para ejecutar este notebook. No modifique esta celda.

```
# Importando las bibliotecas necesarias
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage, signal, misc
from google.colab import files
import imageio as iio
from skimage import img_as_float
import sys, os, time, math
```

4.12 Función para mostrar imágenes

Esta función auxiliar se utiliza para visualizar las imágenes generadas en los ejercicios.

```
def mostrar_imagem(imagem, titulo=''):
    """
    Función para mostrar una imagen en escala de grises.
    """
    plt.figure(figsize=(6,6))
    plt.imshow(imagem, cmap='gray', vmin=0, vmax=1)
    plt.title(titulo)
    plt.axis('off')
    plt.show()
```

4.13 Funciones para agregar ruido

Complete las funciones a continuación para agregar diferentes tipos de ruido a una imagen. Siga las indicaciones de cada sección.

4.13.1 Ruido sal y pimienta

Implemente la función `ruido_salpimienta(imagem, prob)`, que agrega ruido sal y pimienta a una imagen en escala de grises (valores entre 0 y 1). La probabilidad `prob` indica la probabilidad total de que un píxel

sea modificado (siendo la mitad para sal y la mitad para pimienta, en promedio).

```
# EJERCICIO 1: Implemente la función para agregar ruido sal y pimienta
def ruido_salpimienta(imagem, prob):
    """
    Agrega ruido sal y pimienta a una imagen.
    Parámetros:
        imagem: array 2D con valores entre 0 y 1
        prob: probabilidad total de ruido (entre 0 y 1)
    Retorna:
        imagen con ruido
    """
    saida = imagem.copy()
    # Complete el código aquí
    # 1. Genere una matriz de números aleatorios uniformes en [0, 1)
    #    con el mismo tamaño que la imagen.
    # 2. Los píxeles con valor < prob/2 deben convertirse en 0 (pimienta).
    # 3. Los píxeles con valor entre prob/2 y prob deben convertirse en 1 (sal).
    # 4. Los demás píxeles permanecen igual.

    # <<< Inserte aquí su código >>>
    return saida
```

4.13.2 Ruido gaussiano

Implemente la función `ruido_gaussiano(imagem, media=0, sigma=0.05)`, que agrega ruido gaussiano a la imagen. El ruido debe generarse con la función `np.random.normal` y sumarse a la imagen. Asegúrese de que los valores resultantes estén en el rango [0, 1].

```
# EJERCICIO 2: Implemente la función para agregar ruido gaussiano
def ruido_gaussiano(imagem, media=0, sigma=0.05):
    """
    Agrega ruido gaussiano a una imagen.
    Parámetros:
        imagem: array 2D con valores entre 0 y 1
        media: media del ruido
        sigma: desviación estándar del ruido
    Retorna:
        imagen con ruido
    """
    saida = imagem.copy()
    # Complete el código aquí
    # 1. Genere el ruido con np.random.normal(media, sigma, imagem.shape)
    # 2. Sume el ruido a la imagen
    # 3. Recorte los valores para que estén en el rango [0, 1]

    # <<< Inserte aquí su código >>>
    return saida
```

4.13.3 Ruido uniforme

Implemente la función `ruido_uniforme(imagem, intensidade=0.1)`, que agrega ruido con distribución uniforme en el intervalo `[-intensidade, intensidade]` a la imagen. Asegúrese de que los valores resultantes estén en el rango [0, 1].

```
# EJERCICIO 3: Implemente la función para agregar ruido uniforme
def ruido_uniforme(imagem, intensidade=0.1):
    """
    Agrega ruido uniforme a una imagen.
```

```
    Parámetros:
        imagem: array 2D con valores entre 0 y 1
        intensidade: límite del ruido (el ruido se distribuye en [-intensidade, intensidade])
    Retorna:
        imagen con ruido
    """
    saida = imagem.copy()
    # Complete el código aquí
    # 1. Genere el ruido con np.random.uniform(-intensidade, intensidade, imagem.shape)
    # 2. Sume el ruido a la imagen
    # 3. Recorte los valores para que estén en el rango [0, 1]

    # <<< Inserte aquí su código >>>
    return saida
```

4.14 Funciones para filtros

4.14.1 Filtro de media

Implemente la función `filtro_media(imagem, tamanho=3)`, que aplica un filtro de media (promedio) con una máscara de tamaño `tamanho x tamanho` (tamaño impar). Utilice la función `signal.convolve2d` o `ndimage.uniform_filter` de SciPy.

```
# EJERCICIO 4: Implemente el filtro de media
def filtro_media(imagem, tamanho=3):
    """
    Aplica un filtro de media (promedio) a la imagen.
    Parámetros:
        imagem: array 2D
        tamanho: tamaño de la máscara (impar)
    Retorna:
        imagen filtrada
    """
    # Complete el código aquí
    # 1. Defina la máscara (kernel) como una matriz de unos de tamaño (tamanho, tamanho)
    dividida por tamanho**2
    # 2. Aplique la convolución 2D con signal.convolve2d(imagem, mascara, mode='same')
    #    o use ndimage.uniform_filter(imagem, size=tamanho)

    # <<< Inserte aquí su código >>>
    return imagem
```

4.14.2 Filtro de mediana

Implemente la función `filtro_mediana(imagem, tamanho=3)`, que aplica un filtro de mediana con una ventana de tamaño `tamanho x tamanho` (tamaño impar). Utilice `ndimage.median_filter`.

```
# EJERCICIO 5: Implemente el filtro de mediana
def filtro_mediana(imagem, tamanho=3):
    """
    Aplica un filtro de mediana a la imagen.
    Parámetros:
        imagem: array 2D
        tamanho: tamaño de la ventana (impar)
    Retorna:
        imagen filtrada
    """
    # Complete el código aquí
    # 1. Aplique ndimage.median_filter(imagem, size=tamanho)
```

```
    # <<< Inserte aquí su código >>>
    return imagem
```

4.14.3 Filtro gaussiano

Implemente la función `filtro_gaussiano(imagem, sigma=1.0)`, que aplica un filtro gaussiano con desviación estándar `sigma`. Utilice `ndimage.gaussian_filter`.

```
# EJERCICIO 6: Implemente el filtro gaussiano
def filtro_gaussiano(imagem, sigma=1.0):
    """
    Aplica un filtro gaussiano a la imagen.
    Parámetros:
        imagem: array 2D
        sigma: desviación estándar del núcleo gaussiano
    Retorna:
        imagen filtrada
    """
    # Complete el código aquí
    # 1. Aplique ndimage.gaussian_filter(imagem, sigma=sigma)

    # <<< Inserte aquí su código >>>
    return imagem
```

4.15 Carga de la imagen

Ahora cargaremos la imagen que se utilizará en los ejercicios. La imagen debe estar en escala de grises con valores en el rango [0, 1].

```
# Cargando la imagen (elija una de las opciones)
# Opción 1: usar una imagen de la biblioteca scikit-image
from skimage import data
imagem_original = img_as_float(data.camera())

# Opción 2: subir una imagen desde su computadora (descomente el código)
# from google.colab import files
# uploaded = files.upload()
# for filename in uploaded.keys():
#     imagem_original = img_as_float(iio.imread(filename, mode='L'))

# Opción 3: usar imagen generada (descomente el código)
# x, y = np.meshgrid(np.linspace(0, 1, 256), np.linspace(0, 1, 256))
# imagem_original = 0.5 + 0.5 * np.sin(2 * np.pi * (x + y))

print('Tamaño de la imagen:', imagem_original.shape)
print('Rango de valores:', imagem_original.min(), '-', imagem_original.max())
print('Tipo de datos:', imagem_original.dtype)

mostrar_imagem(imagem_original, 'Imagen Original')
```

4.16 Aplicación de ruido

En esta sección, aplicaremos los tres tipos de ruido implementados anteriormente a la imagen original.

```
# Aplicando ruido sal y pimienta
probabilidade = 0.05  # 5% de ruido
imagem_salpimenta = ruido_salpimienta(imagem_original, probabilidade)
```

```
mostrar_imagem(imagem_salpimenta, f'Ruido Sal y Pimienta - Prob: {probabilidade}')
```

```
# Aplicando ruido gaussiano
sigma_ruido = 0.1
imagem_gaussiana = ruido_gaussiano(imagem_original, media=0, sigma=sigma_ruido)
mostrar_imagem(imagem_gaussiana, f'Ruido Gaussiano - Sigma: {sigma_ruido}')
```

```
# Aplicando ruido uniforme
intensidade_ruido = 0.2
imagem_uniforme = ruido_uniforme(imagem_original, intensidade=intensidade_ruido)
mostrar_imagem(imagem_uniforme, f'Ruido Uniforme - Intensidad: {intensidade_ruido}')
```

4.17 Aplicación de filtros sobre imágenes ruidosas

Ahora aplicaremos los filtros implementados a las imágenes ruidosas. Compare los resultados.

4.17.1 Filtro de media sobre diferentes ruidos

```
# Filtro de media sobre ruido sal y pimienta
tamanho_filtro = 3
img_filtrada = filtro_media(imagem_salpimenta, tamanho=tamanho_filtro)
mostrar_imagem(img_filtrada, f'Media (t={tamanho_filtro}) sobre Sal y Pimienta')
```

```
# Filtro de media sobre ruido gaussiano
img_filtrada = filtro_media(imagem_gaussiana, tamanho=tamanho_filtro)
mostrar_imagem(img_filtrada, f'Media (t={tamanho_filtro}) sobre Gaussiano')
```

```
# Filtro de media sobre ruido uniforme
img_filtrada = filtro_media(imagem_uniforme, tamanho=tamanho_filtro)
mostrar_imagem(img_filtrada, f'Media (t={tamanho_filtro}) sobre Uniforme')
```

4.17.2 Filtro de mediana sobre diferentes ruidos

```
# Filtro de mediana sobre ruido sal y pimienta
tamanho_filtro = 3
img_filtrada = filtro_mediana(imagem_salpimenta, tamanho=tamanho_filtro)
mostrar_imagem(img_filtrada, f'Mediana (t={tamanho_filtro}) sobre Sal y Pimienta')
```

```
# Filtro de mediana sobre ruido gaussiano
img_filtrada = filtro_mediana(imagem_gaussiana, tamanho=tamanho_filtro)
mostrar_imagem(img_filtrada, f'Mediana (t={tamanho_filtro}) sobre Gaussiano')
```

```
# Filtro de mediana sobre ruido uniforme
img_filtrada = filtro_mediana(imagem_uniforme, tamanho=tamanho_filtro)
mostrar_imagem(img_filtrada, f'Mediana (t={tamanho_filtro}) sobre Uniforme')
```

4.17.3 Filtro gaussiano sobre diferentes ruidos

```
# Filtro gaussiano sobre ruido sal y pimienta
sigma_filtro = 1.0
img_filtrada = filtro_gaussiano(imagem_salpimenta, sigma=sigma_filtro)
mostrar_imagem(img_filtrada, f'Gaussiano (sigma={sigma_filtro}) sobre Sal y Pimienta')
```

```
# Filtro gaussiano sobre ruido gaussiano
img_filtrada = filtro_gaussiano(imagem_gaussiana, sigma=sigma_filtro)
```

```
mostrar_imagem(img_filtrada, f'Gaussiano (sigma={sigma_filtro}) sobre Gaussiano')
```

```
# Filtro gaussiano sobre ruido uniforme
img_filtrada = filtro_gaussiano(imagem_uniforme, sigma=sigma_filtro)
mostrar_imagem(img_filtrada, f'Gaussiano (sigma={sigma_filtro}) sobre Uniforme')
```

4.18 Comparación de resultados

La celda a continuación genera una figura comparativa con varias combinaciones de ruido/filtro. Ejecútela para visualizar todos los resultados en una sola imagen.

```
# Generando una cuadrícula comparativa de ruidos y filtros
ruidos = [
    ('Original', imagem_original),
    ('Sal y Pimienta', imagem_salpimenta),
    ('Gaussiano', imagem_gaussiana),
    ('Uniforme', imagem_uniforme)
]

filtros = [
    ('Sin Filtro', lambda img: img),
    ('Media', lambda img: filtro_media(img, tamanho=3)),
    ('Mediana', lambda img: filtro_mediana(img, tamanho=3)),
    ('Gaussiano', lambda img: filtro_gaussiano(img, sigma=1.0))
]

fig, axes = plt.subplots(len(ruidos), len(filtros), figsize=(12, 12))

for i, (nome_ruido, img_ruidosa) in enumerate(ruidos):
    for j, (nome_filtro, filtro) in enumerate(filtros):
        resultado = filtro(img_ruidosa)
        axes[i, j].imshow(resultado, cmap='gray', vmin=0, vmax=1)
        if i == 0:
            axes[i, j].set_title(nome_filtro)
        if j == 0:
            axes[i, j].set_ylabel(nome_ruido, fontsize=12)
        axes[i, j].axis('off')

plt.tight_layout()
plt.show()
```

4.19 Ejercicio de análisis

EJERCICIO 7 (Análisis cualitativo): Después de ejecutar las celdas anteriores, responda en la celda Markdown a continuación (haga doble clic para editar):

1. **¿Cuál filtro funciona mejor para eliminar el ruido sal y pimienta? ¿Por qué?**
2. **¿Cuál filtro es más eficaz para reducir el ruido gaussiano? Explique.**
3. **Al aumentar el tamaño de la máscara en el filtro de media, ¿qué efecto observa

4.20 💻 Parte Práctica con Ejercicios de Programación

Esta lista transforma los conceptos del Capítulo 4 en una ruta práctica de segmentación y morfología matemática. Los EPs comienzan con umbralización y avanzan hasta etiquetado y descriptores de componentes, siempre con matrices pequeñas para que cada píxel pueda verificarse a mano.

❗ Regla común de los EPs morfológicos

En las operaciones con vecindad, **no haga padding**. Para cada píxel, evalúe únicamente las posiciones del elemento estructurante que caen dentro del dominio de la imagen. Esta es la misma idea de las implementaciones didácticas en `morph.py`, como `mm.dil0`, `mm.ero0`, `mm.dil1` y `mm.label0`: la vecindad se recorta por el dominio válido de la imagen.

🎯 Objetivo de este cuaderno

El cuaderno permite desarrollar, validar, organizar y probar soluciones de **Ejercicios de Programación (EPs)** en entornos interactivos, como Colab, con los mismos casos de prueba de Moodle, copiándolos allí solo al momento de registrar la nota oficial.

Download

Descargue `morph.py` y `testsuite.py` ejecutando la celda a continuación:

```
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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Ejecutando las pruebas

Para evaluar las pruebas, ejecute `TestSuite("EP04_01.extensión").run()` en una nueva celda, reemplazando la extensión por la del lenguaje utilizado (`.py`, `.java`, `.c`, `.cpp`, `.js` o `.r`). El sistema descarga los casos de prueba de GitHub, ejecuta el programa y calcula la nota automáticamente.

Para probar código Python directamente, sin guardar archivo, use `run_code(codigo)` pasando el código como *string* en una variable `codigo`:

```
codigo = """
from morph import mm
# ... su código aquí ...
"""
TestSuite("EP04_01").run_code(codigo)
```

4.20.1 EP04_01 Umbralización Global por Umbral Fijo

En los **escáneres de documentos** y en los **sistemas de lectura de códigos de barras**, la primera etapa del procesamiento siempre consiste en separar lo que es "objeto" (tinta, texto, barras) de lo que es "fondo" (papel, embalaje). La **umbralización global** hace exactamente esto: compara cada píxel con un único umbral T y decide, en tiempo real, si pertenece a la clase clara o a la clase oscura. Es el operador de segmentación más simple — y aun así, está detrás de buena parte de los *pipelines* industriales de inspección visual. Ver en Figura 4.30 una simulación de este EP.

4.20.1.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (filas) y C (columnas).

2. **Umbral:** Leer el entero T (umbral de decisión).
3. **Datos:** Leer los valores enteros de la matriz original fila a fila.
4. **Mapeo:** Para cada píxel p, calcular el nuevo valor mediante la ecuación:

$$p' = \begin{cases} 255, & \text{si } p > T \\ 0, & \text{si } p \leq T \end{cases}$$

5. **Salida:** Mostrar la matriz binarizada con dimensiones $L \times C$.

4.20.1.2 📌 Restricciones Computacionales

- **Binarización:** La salida contiene **solo** los valores 0 o 255.
- **Comparación estricta:** El criterio usa $> T$ (los píxeles iguales a T se convierten en fondo).
- **Tipo:** El resultado final debe ser entero.
- **Observación:** Este EP sigue la convención de OpenCV (`cv2.THRESL_BINARY`): solo los píxeles con valor **mayor que** T se convierten en blancos (`255`); los píxeles con valor **igual a** T permanecen negros (`0`).

4.20.1.3 🧠 Fundamentación Teórica

Parámetro	Tipo	Impacto Visual
T **pequeño**	Entero	La mayoría de los píxeles se vuelven blancos
T **grande**	Entero	La mayoría de los píxeles se vuelven negros
T **bien elegido**	Entero	Separa nítidamente objeto y fondo

4.20.1.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Entero T.
- Líneas siguientes: Elementos enteros de la matriz original.

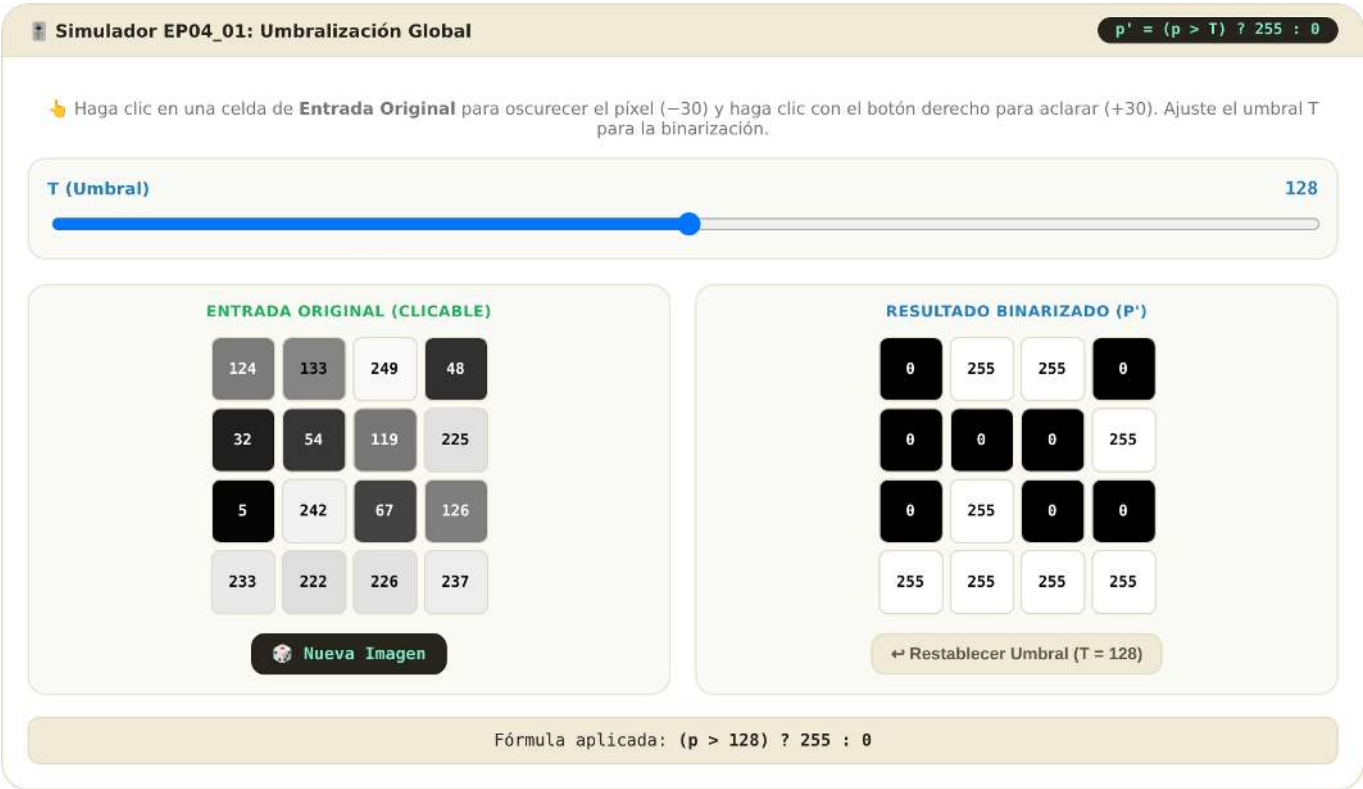
Salida:

- Matriz binarizada en L filas y C columnas, valores 0 o 255 separados por espacio.

4.20.1.5 📌 Ejemplos

Entrada	Salida	Observación
2 4 100 0 99 100 180 255 30 120 80	0 0 0 255 255 0 255 0	$T = 100$: solo los píxeles con valor mayor que 100 se vuelven blancos; por eso, 99 y 100 se vuelven negros.
1 3 0 0 50 255	0 255 255	$T = 0$: solo los píxeles con valor estrictamente mayor que 0 se vuelven blancos.

```
%%writefile EP04_01.py
# Código Python
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-ep0401-limiar.html

Figura 4.30: Simulador EP04_01: Umbralización Global por Umbral Fijo (p' = (p > T) ? 255 : 0)

```
Overwriting EP04_01.py
```

```
TestSuite("EP04_01.py").run()
```

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

4.20.2 EP04_02 📊 Umbralización Automática de Otsu

Elegir manualmente el umbral T funciona cuando la iluminación es estable, pero en **microscopía digital** y en **inspección de láminas de sangre**, cada muestra tiene un contraste diferente — un umbral fijo fallaría de imagen en imagen. El **método de Otsu** resuelve esto encontrando, por sí solo, el umbral que **maximiza la separación estadística** entre las dos clases de píxeles, haciendo la segmentación automática y adaptativa. Ver en Figura 4.31 una simulación de este EP.

4.20.2.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (filas) y C (columnas).
2. **Datos:** Leer los valores enteros de la matriz original fila por fila.
3. **Histograma:** Construir el histograma $h[i]$, $i = 0, \ldots, 255$, contando cuántos píxeles tienen valor i.
4. **Búsqueda del umbral:** Para cada candidato T de 1 a 255, calcular la **varianza entre clases**:

$$\sigma_B^2(T) = \frac{n_0 \cdot n_1}{N^2}(m_0 - m_1)^2$$

   donde n_0, n_1 son las cantidades de píxeles con valor $< T$ y $\geq T$, m_0, m_1 son sus medias, y $N = L \times C$.
5. **Elección:** El umbral óptimo T^* es el que maximiza $\sigma_B^2(T)$ (en caso de empate, mantener el **primero** encontrado).

6. **Aplicación:** Binarizar la imagen usando T^*, aplicando:

$$p' = \begin{cases} 255, & \text{si } p > T^* \\ 0, & \text{si } p \leq T^* \end{cases}$$

4.20.2.2 📌 Restricciones Computacionales

- **Candidatos válidos:** Ignorar T que deje $n_0 = 0$ o $n_1 = 0$ (clase vacía).
- **Empate:** Mantener siempre el **primer** T que alcanzó el valor máximo de σ_B^2.
- **Tipo:** T^* y la matriz de salida deben ser enteros.
- **Convención OpenCV:** La binarización sigue `cv2.THRESL_BINARY`; los píxeles con valor exactamente igual a T^* se vuelven negros.

4.20.2.3 🧠 Fundamentación Teórica

Concepto	Significado	Impacto
$\sigma_B^2(T)$ **alta**	Clases bien separadas en T	T es un buen candidato a umbral
Histograma bimodal	Dos "picos" distintos	Otsu encuentra el valle entre ellos
Histograma unimodal	Un único "pico"	Otsu aún elige *algún* T, pero la segmentación es poco fiable

4.20.2.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Líneas siguientes: Elementos enteros de la matriz original.

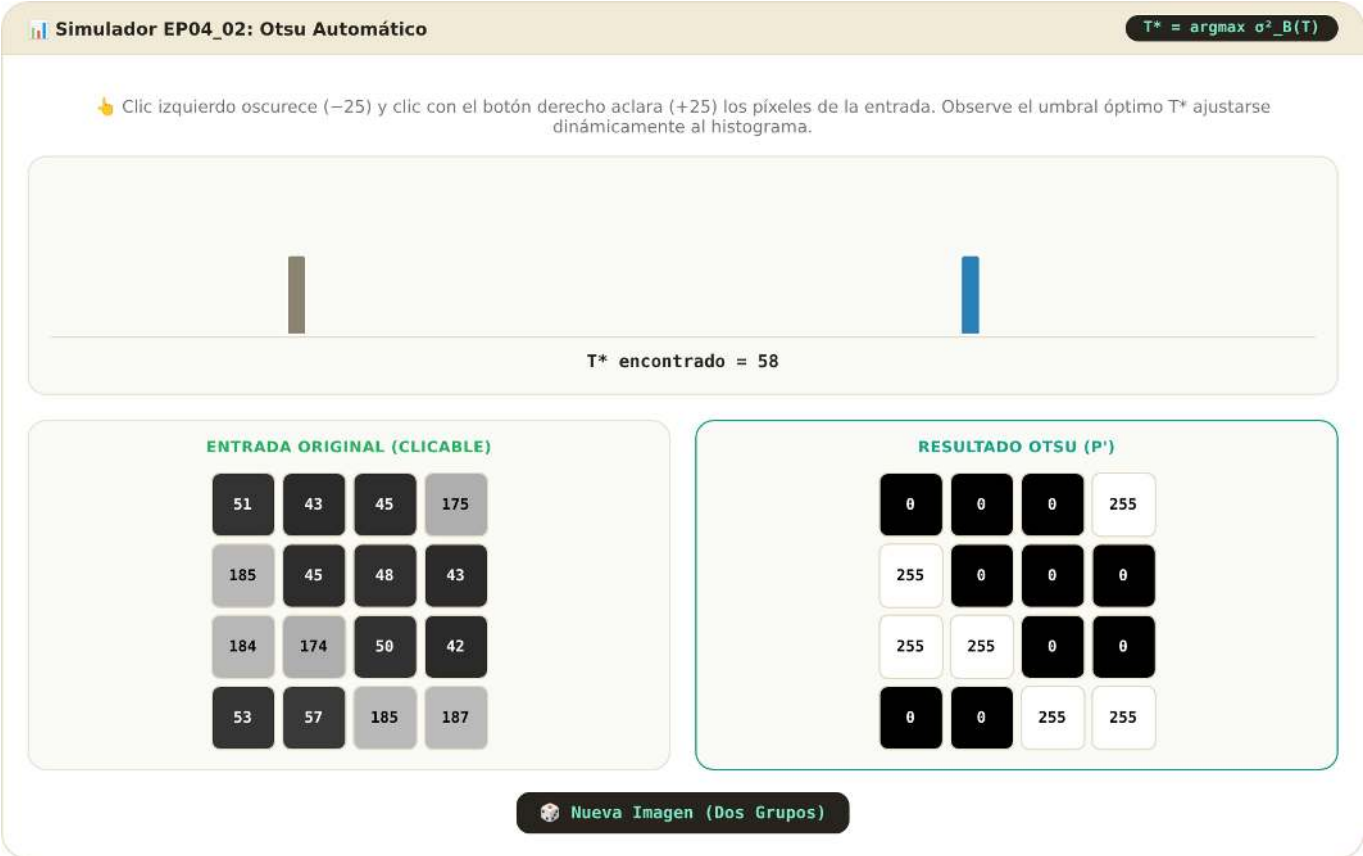
Salida:

- Matriz binarizada en L filas y C columnas, valores 0 o 255.

4.20.2.5 📌 Ejemplos

Entrada	Salida	Observación
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	Histograma bimodal claro: 12 y 200
1 2 10 250	0 250	Solo dos valores: T^* queda en el mayor

```
%%writefile EP04_02.py
# Código Python
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-ep0402-otsu.html

Figura 4.31: Simulador EP04_02: Limiarización Automática de Otsu (T* = argmax ²_B(T))

```
Overwriting EP04_02.py
```

```
TestSuite("EP04_02.py").run()
```

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

4.20.3 EP04_03 🌱 Dilatación Binaria Plana (mm.dil0)

En **microscopía de partículas** y en **OCR de placas de automóvil desgastadas**, los trazos finos o discontinuos deben "engrosarse" para que el reconocimiento funcione. La **dilatación morfológica** hace exactamente eso: expande regiones claras usando un elemento estructurante B — la misma operación implementada en `morph.py` como `mm.dil0(f, B)`, usada cuando B es **plano** (sin pesos, solo 0/1). Ver en Figura 4.32 una simulación de este EP.

4.20.3.1 📋 Directrices de Implementación

1. **Dimensiones de la imagen:** Leer los enteros L (filas) y C (columnas) de f.
2. **Dimensiones de B:** Leer los enteros L_B (filas) y C_B (columnas) del elemento estructurante.
3. **Elemento estructurante:** Leer la matriz B con valores 0 o 1, fila a fila.
4. **Datos:** Leer la matriz f (la imagen original), fila a fila.
5. **Reflexión:** Construir B_{ref}, la versión de B reflejada en 180° (filas y columnas invertidas) — exactamente como hace `mm.dil0` internamente.
6. **Vecindad sin padding:** Para cada píxel (y, x), recorrer las posiciones (by, bx) de B_{ref} centradas en (y, x), usando el desplazamiento

$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\frac{L_B}{2} + 0{,}5, \quad o_x = -\frac{C_B}{2} + 0{,}5$$

Descartar todo (v_y, v_x) fuera de $[0, L) \times [0, C)$ — **no rellenar con ceros**.

7. **Mapeo:** Calcular cada píxel de salida como el **máximo** entre $f(y, x)$ y todos los $f(v_y, v_x)$ válidos cuya posición correspondiente en B_{ref} vale 1:

$$g(y, x) = \max\left(f(y, x), \max_{\substack{(v_y, v_x) \text{ válido} \\ B_{ref}(by, bx)=1}} f(v_y, v_x)\right)$$

8. **Salida:** Mostrar la matriz g con dimensiones $L \times C$.

4.20.3.2 📌 Restricciones Computacionales

- **Sin padding:** Nunca inventar vecinos fuera de la imagen; usar solo los que existen realmente.
- **Reflexión obligatoria:** B debe reflejarse antes de aplicarse (es lo que diferencia `mm.dil0` de una simple búsqueda de máximo).
- **Robustez de borde:** Si ninguna posición válida de $B_{ref} = 1$ cae dentro del dominio para un píxel dado, este **mantiene su valor original**.

4.20.3.3 🧠 Fundamentación Teórica

Concepto	Significado	Impacto Visual
Dilatación	$g \geq f$ siempre (extensiva)	Las regiones claras crecen, los huecos oscuros se encogen
B **mayor**	Vecindad más amplia	Crecimiento más agresivo
Reflexión de B	$B_{ref}(y, x) = B(-y, -x)$	Garantiza la definición formal de Minkowski de la dilatación

4.20.3.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

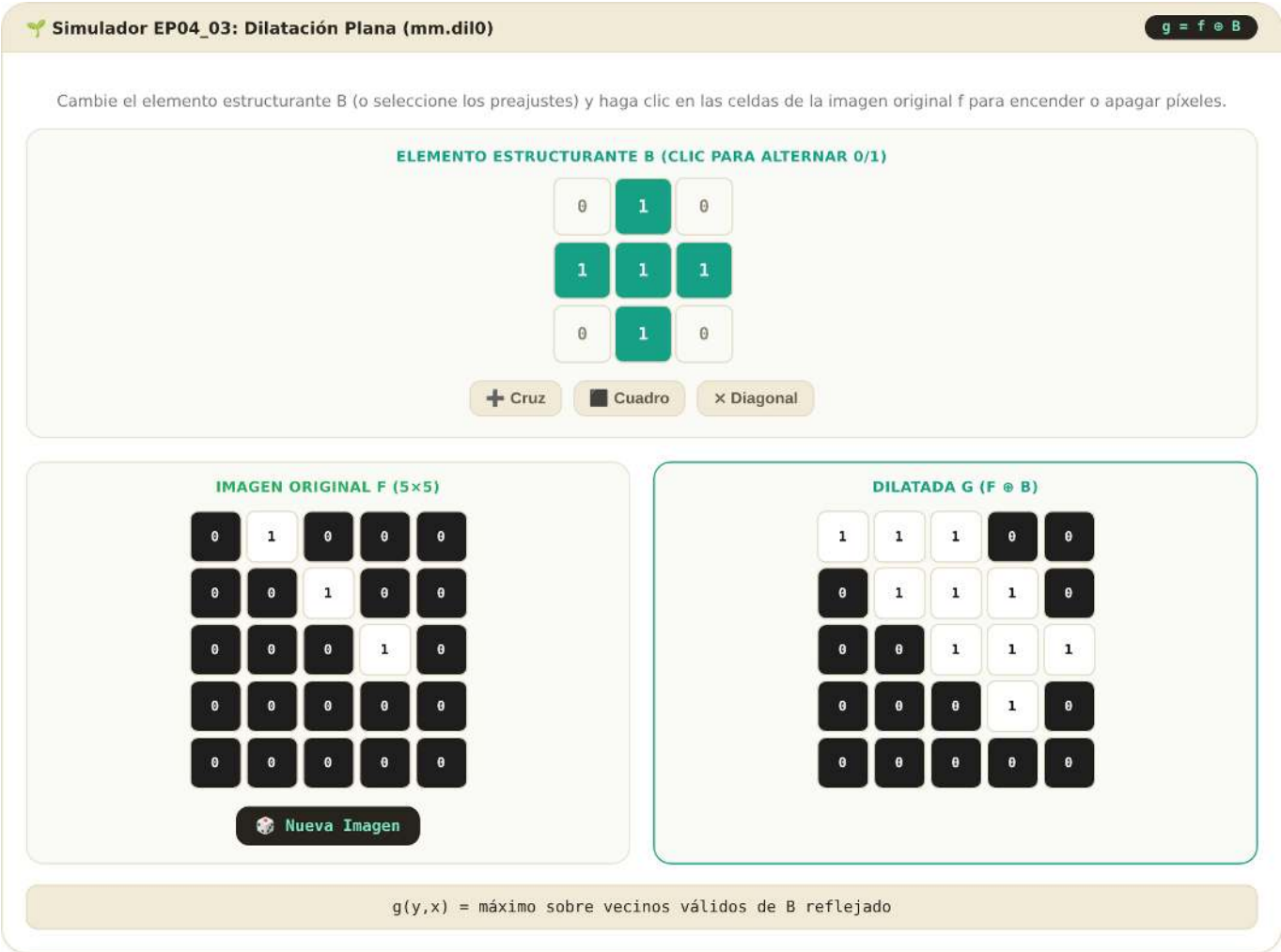
- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Entero L_B.
- Línea 4: Entero C_B.
- Siguientes L_B líneas: elementos enteros (0 o 1) de la matriz B.
- Siguientes L líneas: elementos enteros de la matriz f.

Salida:

- Matriz g en L filas y C columnas, valores enteros separados por espacio.

4.20.3.5 📌 Ejemplos

Entrada	Salida	Observación
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	B en cruz simétrico: punto aislado se expande en cruz
1 4 1 3 1 1 1 10 200 5 80	200 200 200 80	B horizontal: cada píxel "atrae" el máximo de los vecinos de la fila



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-ep0403-dilatacao.html

Figura 4.32: Simulador EP04_03: Dilatación Binaria Plana (g = f B)

```
%%writefile EP04_03.py
# Código Python
```

```
Overwriting EP04_03.py
```

```
TestSuite("EP04_03.py").run()
```

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

4.20.4 EP04_04 🪨 Erosión Binaria Plana (mm.ero0)

Si la dilatación engrosa, la **erosión** afina. En **sistemas de conteo de células**, se utiliza para **separar células que se tocan**: al "comer" los bordes de cada región, las conexiones finas entre objetos desaparecen incluso antes de que se realice cualquier conteo. En `morph.py`, esta es la operación `mm.ero0(f, B)` — la **dual** exacta de la dilatación, y la única de las dos que **no** refleja el elemento estructurante. Ver en Figura 4.33 una simulación de este EP.

4.20.4.1 📋 Directrices de Implementación

1. **Dimensiones de la imagen:** Leer los enteros L (filas) y C (columnas) de f.
2. **Dimensiones de B:** Leer los enteros L_B (filas) y C_B (columnas) del elemento estructurante.
3. **Elemento estructurante:** Leer la matriz B con valores 0 o 1, fila a fila.
4. **Datos:** Leer la matriz f (la imagen original), fila a fila.
5. **Vecindad sin padding (¡sin reflexión!):** Para cada píxel (y, x), recorrer las posiciones (by, bx) de B **en el orden original** (sin reflejar), usando el mismo desplazamiento del EP04_03:

$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\tfrac{L_B}{2} + 0{,}5, \quad o_x = -\tfrac{C_B}{2} + 0{,}5$$

Descartar todo (v_y, v_x) fuera de $[0, L) \times [0, C)$. 6. **Mapeo:** Calcular cada píxel de salida como el **mínimo** entre $f(y, x)$ y todos los $f(v_y, v_x)$ válidos cuya posición correspondiente en B vale 1:

$$g(y, x) = \min\Big(f(y, x), \min_{\substack{(v_y, v_x) \text{ válido} \\ B(by, bx)=1}} f(v_y, v_x)\Big)$$

7. **Salida:** Mostrar la matriz g con dimensiones $L \times C$.

4.20.4.2 📌 Restricciones Computacionales

- **Sin reflexión:** A diferencia de la dilatación, B se usa **exactamente como se lee** — reflejarlo aquí sería un error conceptual grave.
- **Sin padding:** Los vecinos fuera de la imagen simplemente se ignoran, nunca se tratan como 0.
- **Robustez de borde:** Si ninguna posición válida de $B = 1$ cae dentro del dominio, el píxel mantiene su valor original.

4.20.4.3 🧠 Fundamentación Teórica

Concepto	Significado	Impacto Visual
Erosión	$g \leq f$ siempre (anti-extensiva)	Las regiones claras se encogen, el ruido puntual desaparece
Dualidad	$\text{ero}(f, B) = -\text{dil}(-f, B_{ref})$	La erosión y la dilatación son "espejos" matemáticos
B **más grande**	Erosión más agresiva	Los objetos finos desaparecen por completo

4.20.4.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Entero L_B.
- Línea 4: Entero C_B.
- Siguientes L_B líneas: elementos enteros (0 o 1) de la matriz B.
- Siguientes L líneas: elementos enteros de la matriz f.

Salida:

- Matriz g en L filas y C columnas, valores enteros separados por espacios.

4.20.4.5 📌 Ejemplos

Entrada	Salida	Observación
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	El “agujero” central (0) se propaga en cruz
1 4 1 3 1 1 1 10 200 5 80	10 5 5 80	B horizontal: cada píxel “extrae” el mínimo de los vecinos de la fila

```
%%writefile EP04_04.py
# Código Python
```

```
Overwriting EP04_04.py
```

```
TestSuite("EP04_04.py").run()
```

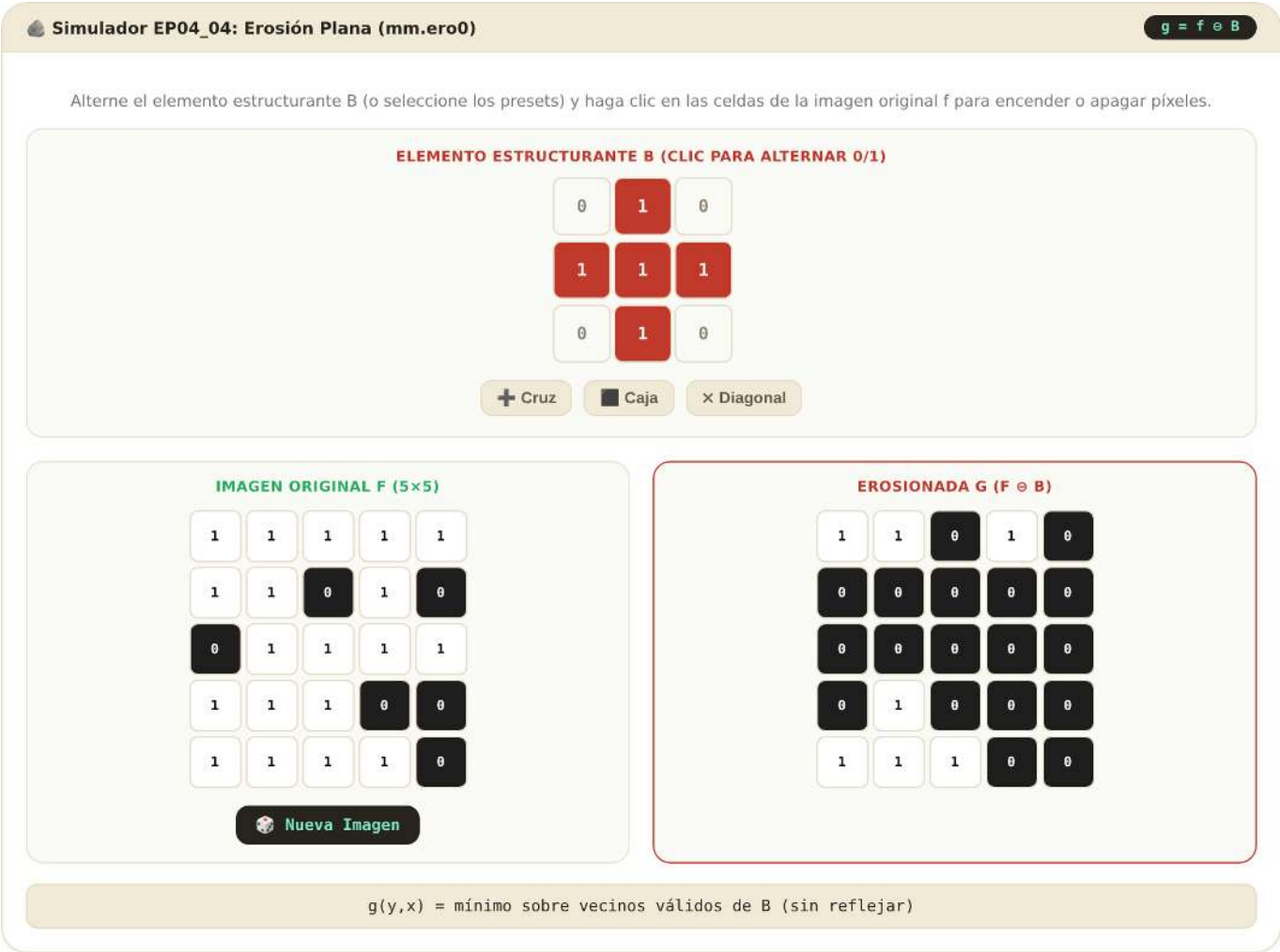
```
<IPython.core.display.HTML object>
```

4.20.5 EP04_05 🧹 Apertura Morfológica (Eliminación de Ruido)

Las imágenes capturadas por **sensores de bajo costo**, como los de drones agrícolas, suelen venir salpicadas de pequeños puntos de ruido — píxeles aislados que no representan nada real. Aplicar erosión seguida de dilatación con el **mismo** elemento estructurante produce la **apertura**: esta “limpia” puntos y protuberancias finas, pero devuelve al objeto principal prácticamente su tamaño original. Es la combinación clásica utilizada en **preprocesamiento de imágenes de satélite** antes de cualquier conteo de área plantada. Ver en Figura 4.34 una simulación de este EP.

4.20.5.1 📋 Directrices de Implementación

1. **Dimensiones de la imagen:** Leer los enteros L (filas) y C (columnas) de f.
2. **Dimensiones de B:** Leer los enteros L_B (filas) y C_B (columnas) del elemento estructurante.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-ep0404-erosao.html

Figura 4.33: Simulador EP04_04: Erosión Binaria Plana (g = f B)

3. **Elemento estructurante:** Leer la matriz B con valores 0 o 1, fila a fila.
4. **Datos:** Leer la matriz binaria f (valores 0 o 1), fila a fila.
5. **Erosión:** Calcular $e = f \ominus B$, usando exactamente el algoritmo del EP04_04 (sin reflejar B, sin padding).
6. **Dilatación:** Calcular $g = e \oplus B$, usando exactamente el algoritmo del EP04_03 (reflejando B, sin padding) — pero ahora aplicado sobre e, no sobre f.
7. **Salida:** Mostrar la matriz resultante g (la **apertura** de f por B) con dimensiones $L \times C$.

4.20.5.2 📌 Restricciones Computacionales

- **Orden fijo:** Es **siempre** erosión primero, luego dilatación — el orden inverso define otro operador (cierre, del próximo EP).
- **Mismo B:** El elemento estructurante utilizado en la erosión y en la dilatación debe ser idéntico.
- **Sin padding en ninguna de las dos etapas.**

4.20.5.3 🧠 Fundamentación Teórica

Concepto	Significado	Impacto Visual
Antiextensividad	$g \subseteq f$ siempre	La apertura nunca crea un píxel nuevo, solo elimina
Idempotencia	apertura(apertura(f)) = apertura(f)	Aplicar de nuevo no cambia nada más
Puntos aislados	Menores que B	Son completamente eliminados
Núcleo del objeto	Mayor que B	Se recupera casi intacto mediante la dilatación final

4.20.5.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

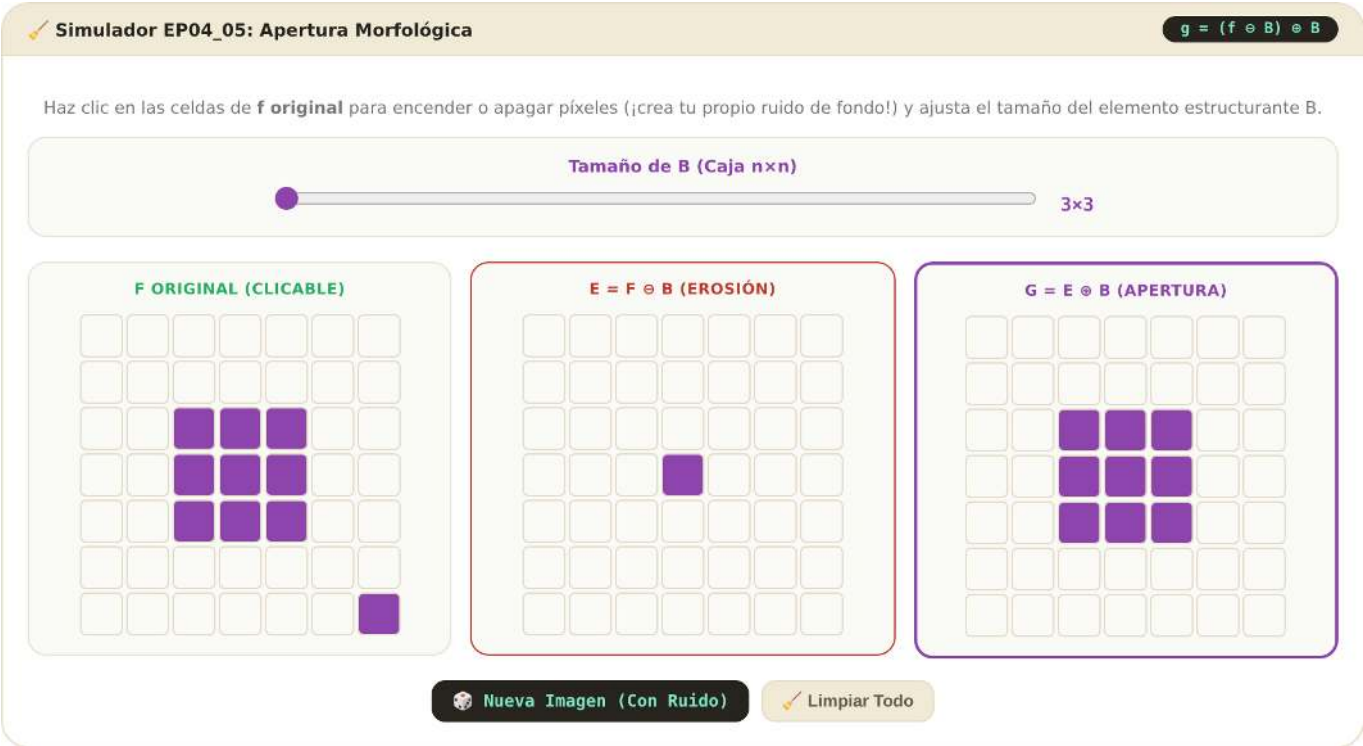
- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Entero L_B.
- Línea 4: Entero C_B.
- Siguientes L_B líneas: elementos enteros (0 o 1) de la matriz B.
- Siguientes L líneas: elementos enteros (0 o 1) de la matriz f.

Salida:

- Matriz resultante en L filas y C columnas, valores 0 o 1.

4.20.5.5 📌 Ejemplos

Entrada	Salida	Observación
7 7 3 3 1 1 1 1 1 1 1 1 1 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	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1 1 1 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Los puntos aislados y la protuberancia fina desaparecen; el cuadrado central sobrevive



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-ep0405-abertura.html

Figura 4.34: Simulador EP04_05: Apertura Morfológica (g = (f B) B)

```
%%writefile EP04_05.py
# Código Python
```

```
Overwriting EP04_05.py
```

```
TestSuite("EP04_05.py").run()
```

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

4.20.6 EP04_06 🧩 Cierre Morfológico (Relleno de Huecos)

En la **digitalización de huellas dactilares**, los surcos de la piel a veces se ven interrumpidos por suciedad o sequedad, creando pequeñas fallas en la curva continua que debería existir. El **cierre** —dilatación seguida de erosión con el mismo elemento estructurante— es el operador dual de la apertura: **rellena huecos pequeños y entrantes estrechos**, sin alterar significativamente el contorno externo del objeto. Es el paso estándar antes de extraer el esqueleto de una huella dactilar. Ver en Figura 4.35 una simulación de este EP.

4.20.6.1 📋 Directrices de Implementación

1. **Dimensiones de la imagen:** Leer los enteros L (líneas) y C (columnas) de f.
2. **Dimensiones de B:** Leer los enteros L_B (líneas) y C_B (columnas) del elemento estructurante.
3. **Elemento estructurante:** Leer la matriz B con valores 0 o 1, línea por línea.
4. **Datos:** Leer la matriz binaria f (valores 0 o 1), línea por línea.
5. **Dilatación:** Calcular $d = f \oplus B$, usando exactamente el algoritmo del EP04_03 (reflejando B, sin padding).
6. **Erosión:** Calcular $g = d \ominus B$, usando exactamente el algoritmo del EP04_04 (sin reflejar B, sin padding) — ahora aplicado sobre d, no sobre f.
7. **Salida:** Mostrar la matriz resultante g (el **cierre** de f por B) con dimensiones $L \times C$.

4.20.6.2 📌 Restricciones Computacionales

- **Orden fijo:** Es **siempre** dilatación primero, luego erosión — el orden inverso es la apertura del EP04_05.
- **Mismo B:** El elemento estructurante usado en la dilatación y en la erosión debe ser idéntico.
- **Sin padding en ninguna de las dos etapas.**

4.20.6.3 🧠 Fundamentación Teórica

Concepto	Significado	Impacto Visual
Extensividad	$g \supseteq f$ siempre	El cierre nunca elimina píxeles, solo añade
Idempotencia	$\text{cierre}(\text{cierre}(f)) = \text{cierre}(f)$	Aplicarlo de nuevo no cambia nada más
Huecos pequeños	Menores que B	Se rellenan completamente
Dualidad	$\text{cierre}(f) = \overline{\text{apertura}(\bar{f})}$	Es la apertura aplicada al "negativo" de la imagen

4.20.6.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

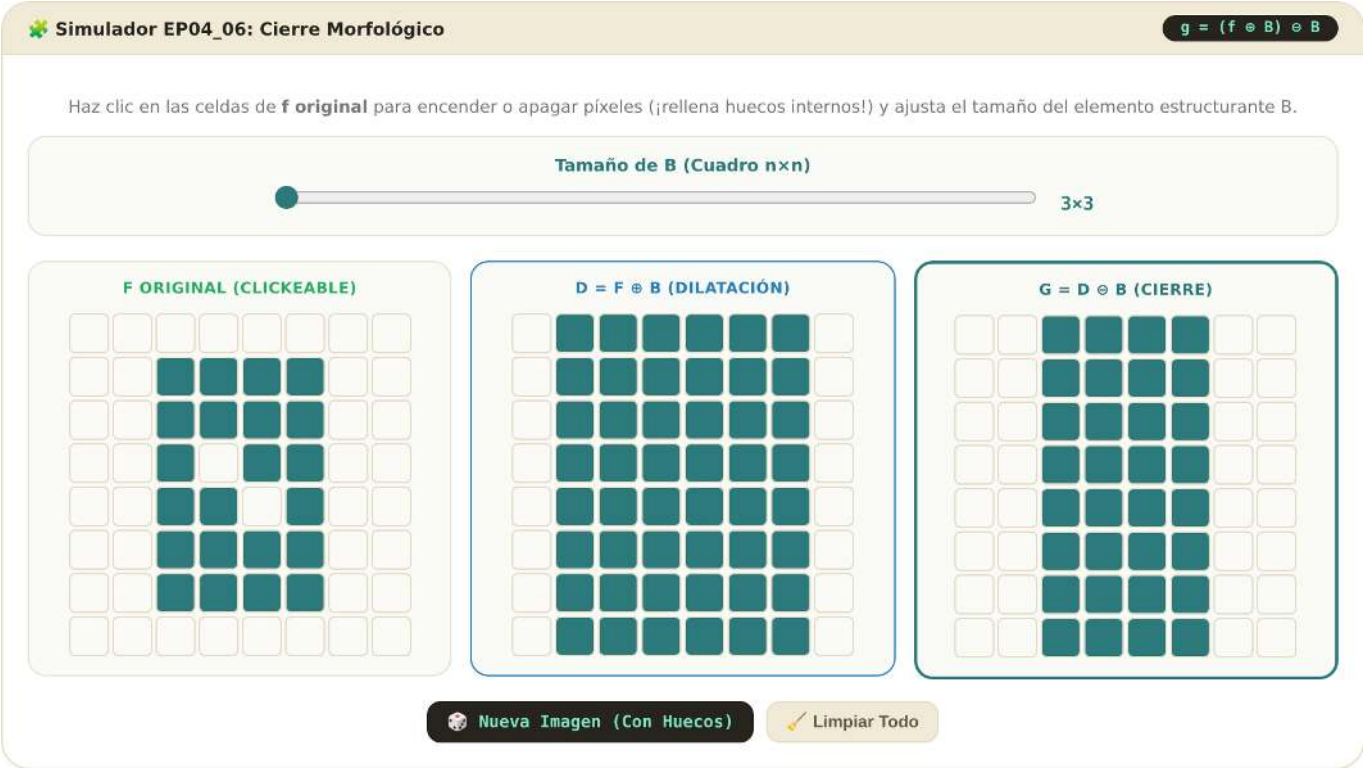
- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Entero L_B.
- Línea 4: Entero C_B.
- Siguientes L_B líneas: elementos enteros (0 o 1) de la matriz B.
- Siguientes L líneas: elementos enteros (0 o 1) de la matriz f.

Salida:

- Matriz resultante en L líneas y C columnas, valores 0 o 1.

4.20.6.5 📌 Ejemplos

Entrada	Salida	Observación
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	Los dos huecos internos no adyacentes se rellenan totalmente



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-ep0406-fechamento.html

Figura 4.35: Simulador EP04_06: Cierre Morfológico (g = (f B) B)

```
%%writefile EP04_06.py
# Código Python
```

```
Overwriting EP04_06.py
```

```
TestSuite("EP04_06.py").run()
```

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

4.20.7 EP04_07 Dilatación y Erosión con Pesos (mm.dil1 / mm.ero1)

Hasta ahora, el elemento estructurante solo decía "este vecino cuenta" o "no cuenta" — pero en **modelos digitales de elevación** (usados en SIG y en planificación de drenaje urbano), cada vecino debería tener un **peso diferente** dependiendo de la distancia o de la dirección del relieve. Las versiones **ponderadas** de la dilatación y de la erosión, implementadas en `morph.py` como `mm.dil1(f, b)` y `mm.ero1(f, b)`, suman (o restan) el peso de cada vecino antes de tomar el máximo (o mínimo) — generalizando todo lo realizado en los EPs anteriores. Ver en Figura 4.36 una simulación de este EP.

4.20.7.1 📋 Directrices de Implementación

1. **Dimensiones de la imagen:** Leer los enteros L (filas) y C (columnas) de f.
2. **Dimensiones de b:** Leer los enteros L_B (filas) y C_B (columnas) del elemento estructurante ponderado.
3. **Pesos:** Leer la matriz b de pesos **enteros** (pueden ser negativos, cero o positivos), fila a fila.
4. **Datos:** Leer la matriz f (la imagen original), fila a fila.
5. **Vecindario sin padding:** Para cada píxel (y, x), recorrer **todas** las posiciones (by, bx) de b (no solo donde valdría 1 — aquí **todo** peso participa), usando el mismo desplazamiento de los EPs anteriores:

$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\tfrac{L_B}{2} + 0{,}5, \quad o_x = -\tfrac{C_B}{2} + 0{,}5$$

   Descartar todo (v_y, v_x) fuera de $[0, L) \times [0, C)$.
6. **Dilatación ponderada:** Calcular

$$g_{dil}(y, x) = \max\Big(f(y, x), \max_{(v_y, v_x)\ \text{válido}} \big(f(v_y, v_x) + b(by, bx)\big)\Big)$$

7. **Erosión ponderada:** Calcular, **usando el mismo b y sin reflejar**:

$$g_{ero}(y, x) = \min\Big(f(y, x), \min_{(v_y, v_x)\ \text{válido}} \big(f(v_y, v_x) - b(by, bx)\big)\Big)$$

8. **Salida:** Mostrar **primero** la matriz g_{dil} completa, y **después** la matriz g_{ero} completa.

4.20.7.2 📌 Restricciones Computacionales

- **Ninguna de las dos refleja** b — la versión ponderada no usa reflexión, incluso en la dilatación (diferente de `mm.dil0`).
- **Todos los pesos participan:** No existe aquí el filtro "$B = 1$"; incluso el peso 0 entra en la cuenta.
- **Sin padding:** los vecinos fuera de la imagen se ignoran, nunca se rellenan virtualmente.
- **Tipo:** La salida puede contener valores negativos o mayores que 255 — **no** hay *clipping* en este EP.
- **Consejo:** Para eliminar mensajes de desbordamiento al superar los límites del tipo uint8, incluir al inicio del código:

```
import warnings
warnings.filterwarnings("ignore")
```

4.20.7.3 🧠 Fundamentación Teórica

Concepto	Significado	Impacto Visual
Peso positivo	"Empuja" el valor del vecino hacia arriba en la dilatación	Simula relieve que asciende en esa dirección
Peso negativo	Reduce la contribución del vecino	Simula distancia o atenuación direccional
Dualidad ponderada	$\text{ero1}(f, b) = -\text{dil1}(-f, b)$	La simetría entre las dos operaciones se mantiene incluso con pesos

4.20.7.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Entero L_B.
- Línea 4: Entero C_B.
- Siguientes L_B líneas: elementos enteros (pueden ser negativos) de la matriz b.
- Siguientes L líneas: elementos enteros de la matriz f.

Salida:

- Primero la matriz g_{dil} en L líneas y C columnas.
- A continuación la matriz g_{ero} en L líneas y C columnas.

4.20.7.5 📌 Ejemplos

Entrada	Salida	Observación
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	Peso central 2 acelera el crecimiento en la dilatación y la contracción en la erosión

```
%%writefile EP04_07.py
# Código Python
```

```
Overwriting EP04_07.py
```

```
TestSuite("EP04_07.py").run()
```

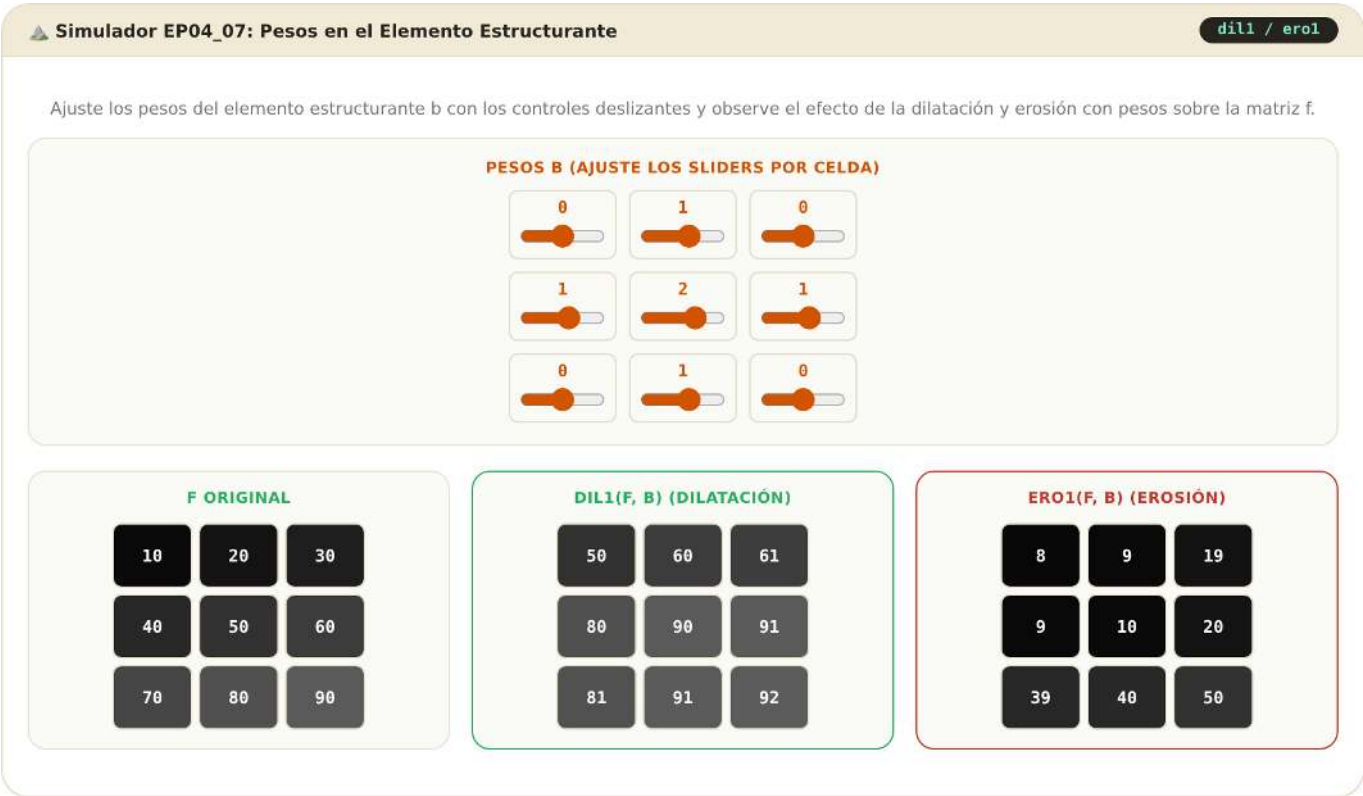
```
<IPython.core.display.HTML object>
```

4.20.8 EP04_08 🌋 Gradiente morfológico, Top-hat y Black-hat

En la **inspección automática de placas de circuito**, tres preguntas aparecen todo el tiempo: ¿dónde están los **bordes** de los componentes? ¿Qué **detalles claros y pequeños** (como puntos de soldadura) se destacan del fondo? ¿Qué **reentrancias oscuras** (como fisuras) esconde el fondo? Un único par erosión/dilatación responde a las tres: el **gradiente morfológico** evidencia contornos, el **top-hat** revela picos estrechos, y el **black-hat** revela valles estrechos — tres herramientas, una sola vecindad. Ver en Figura 4.37 una simulación de este EP.

4.20.8.1 📋 Directrices de Implementación

1. **Dimensiones de la imagen:** Leer los enteros L (filas) y C (columnas) de f.
2. **Dimensiones de B:** Leer los enteros L_B (filas) y C_B (columnas) del elemento estructurante.
3. **Elemento estructurante:** Leer la matriz B con valores 0 o 1, línea a línea.
4. **Datos:** Leer la matriz f (la imagen original, en tonos de gris), línea a línea.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-ep0407-pesos.html

Figura 4.36: Simulador EP04_07: Dilatación y Erosión con Pesos (mm.dil1 / mm.ero1)

5. **Operadores de base:** Calcular, exactamente como en los EPs 04_03 a 04_06:
   - $d = f \oplus B$ (dilatación),
   - $e = f \ominus B$ (erosión),
   - abertura $= e \oplus B$,
   - fechamento $= d \ominus B$.
6. **Gradiente morfológico:** $\text{grad}(y, x) = d(y, x) - e(y, x)$.
7. **Top-hat:** $\text{tophat}(y, x) = f(y, x) - \text{abertura}(y, x)$.
8. **Black-hat:** $\text{blackhat}(y, x) = \text{fechamento}(y, x) - f(y, x)$.
9. **Salida:** Mostrar, **en este orden**, las tres matrices completas: gradiente, top-hat, black-hat.

4.20.8.2 📌 Restricciones Computacionales

- **Sin padding en ninguna etapa intermedia** — dilatación, erosión, apertura y cierre siguen las mismas reglas de vecindad de los EPs anteriores.
- **No hay *clipping*:** las tres salidas pueden contener cualquier valor entero (el gradiente es siempre ≥ 0, pero top-hat y black-hat también).
- **Reutilización:** d y e deben calcularse **una única vez** y reutilizarse para montar apertura, cierre y gradiente.

4.20.8.3 🧠 Fundamentación Teórica

Operador	Fórmula	Qué revela
Gradiente	$d - e$	Bordes: cero en regiones planas, alto en las transiciones
Top-hat	$f - \text{abertura}(f)$	Elementos **claros y finos**, más pequeños que B

Operador	Fórmula	Qué revela
Black-hat	fechamento$(f) - f$	Elementos **oscuros y finos**, más pequeños que B

4.20.8.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Entero L_B.
- Línea 4: Entero C_B.
- Siguientes L_B líneas: elementos enteros (0 o 1) de la matriz B.
- Siguientes L líneas: elementos enteros de la matriz f.

Salida:

- Matriz gradiente en L filas y C columnas.
- Matriz top-hat en L filas y C columnas.
- Matriz black-hat en L filas y C columnas.

4.20.8.5 📌 Ejemplos

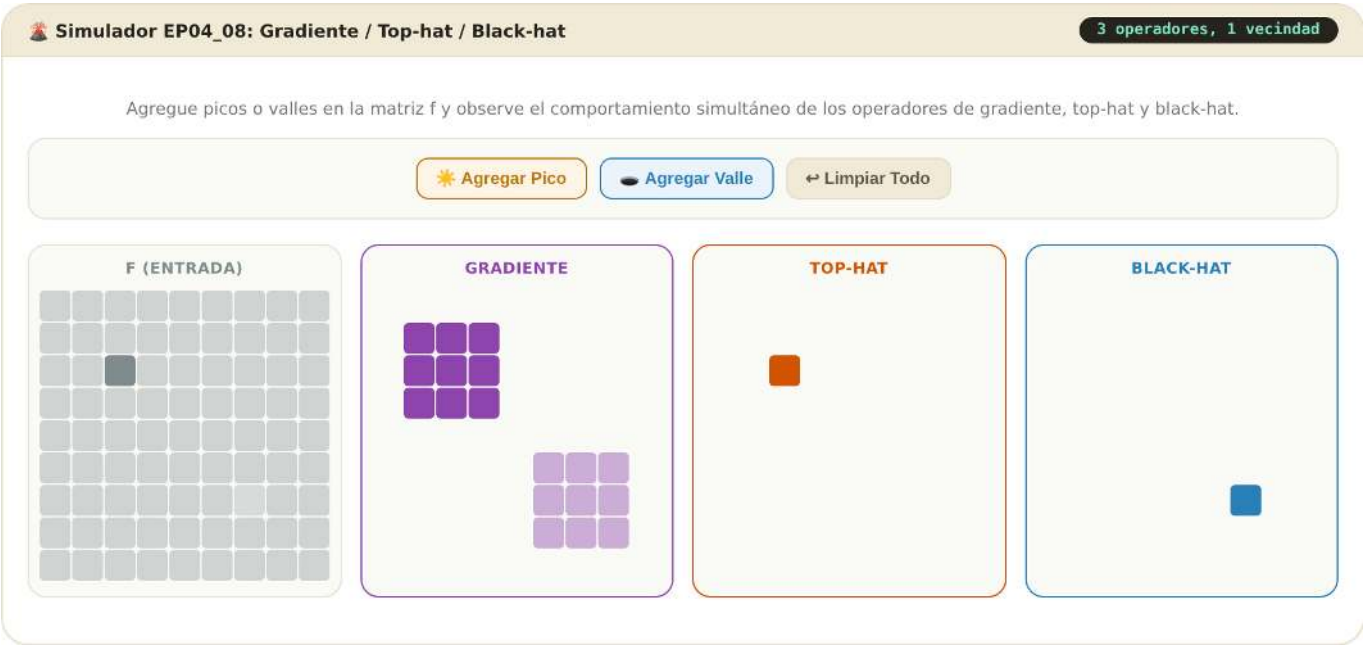
Entrada	Salida	Observación
9 9 3 3 1 1 1 1 1 1 1 1 1 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 80 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 2 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10	(gradiente: halo $3 \times 3 = 70$ en torno a $(2, 2)$ y halo $3 \times 3 = 8$ en torno a $(6, 6)$, resto 0) (top-hat: único 70 en $(2, 2)$, resto 0) (black-hat: único 8 en $(6, 6)$, resto 0)	Pico aislado se convierte en top-hat; valle aislado se convierte en black-hat; ambos aparecen en el gradiente

```
%%writefile EP04_08.py
# Código Python
```

```
Overwriting EP04_08.py
```

```
TestSuite("EP04_08.py").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-ep0408-gradiente.html

Figura 4.37: Simulador EP04_08: Gradiente Morfológico, Top-hat y Black-hat

4.20.9 EP04_09 Transformada de Distancia y el "Centro" del Objeto

En **robótica móvil**, al planificar una ruta dentro de un pasillo, el robot quiere saber no solo *dónde* hay espacio libre, sino también **qué tan lejos** está cada punto libre de la pared más cercana. Las rutas más seguras tienden a pasar por el "centro" del pasillo, lejos de los obstáculos.

La **transformada de distancia morfológica** asigna a cada píxel un valor que representa su distancia hasta el borde más cercano, según la métrica definida por el elemento estructurante. Los píxeles cercanos al borde reciben valores bajos, mientras que los píxeles más internos reciben valores mayores. El píxel de valor máximo corresponde a la región más protegida del objeto, frecuentemente asociada a su centro morfológico.

Ver en Figura 4.38 una simulación de este EP.

4.20.9.1 📋 Directrices de Implementación

1. **Dimensiones de la imagen:** leer los enteros L (filas) y C (columnas) de la imagen f.
2. **Dimensiones de B:** leer los enteros L_B (filas) y C_B (columnas) del elemento estructurante.
3. **Elemento estructurante:** leer la matriz b, que contiene el valor 0 en el centro y valores negativos en las demás posiciones.
4. **Imagen:** leer la matriz binaria f (valores 0 o 1), fila a fila.
5. **Preparación:** multiplicar la imagen por $L \times C$, asegurando que los píxeles internos tengan un valor inicial suficientemente alto para la propagación de las distancias.
6. **Transformada de distancia:** calcular la matriz de distancias utilizando el método `mm.dist1(f,b)`.
7. **Salida:** mostrar la matriz resultante de la transformada de distancia.

4.20.9.2 📌 Restricciones Computacionales

- Utilizar la implementación de erosión ponderada proporcionada por la biblioteca.
- El elemento estructurante puede contener valores negativos arbitrarios.
- La transformada debe obtenerse mediante la aplicación iterativa de erosiones ponderadas hasta alcanzar un punto fijo.

⚠️ **Nota Crucial sobre Lectura de Matrices:** Como el elemento estructurante puede contener valores enteros negativos (por ejemplo, `-1` y `-99`), **no utilice la función `mm.readImg` para leer la matriz** b. Esa función convierte los datos al tipo `uint8`, provocando *underflow* y corrompiendo los valores negativos. Lea las

L_B filas de b manualmente utilizando el tipo estándar `int`. La imagen f puede seguir leyéndose normalmente con `mm.readImg`.

4.20.9.3 🧠 Fundamentación Teórica

Concepto	Significado	Impacto Visual
dist(y, x)	Distancia morfológica hasta el borde más cercano según la métrica definida por b	Los píxeles más internos reciben valores mayores
Valor máximo	Píxel más distante del borde	Aproxima el centro morfológico del objeto
Elemento estructurante ponderado	Define los costos de desplazamiento entre píxeles vecinos	Determina la métrica de distancia utilizada
Objetos finos	Regiones estrechas del objeto	Producen valores bajos de distancia

4.20.9.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: entero L.
- Línea 2: entero C.
- Línea 3: entero L_B.
- Línea 4: entero C_B.
- Siguientes L_B líneas: elementos enteros de la matriz b.
- Siguientes L líneas: elementos binarios (0 o 1) de la matriz f.

⚠️ **Nota de implementación:** Los elementos de la matriz f (0 o 1) deben multiplicarse por **255** para generar una imagen binaria adecuada (0 y 255) antes de aplicar la Transformada de Distancia (TD).

Salida:

- Matriz de la transformada de distancia en L filas y C columnas.

4.20.9.5 📌 Ejemplo

Entrada	Salida	Observación
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	Resultado de la transformada de distancia.

Nota: el valor `-99` actúa como una aproximación práctica de $-\infty$, impidiendo la propagación por las diagonales. De esta forma, solo los vecinos horizontal y vertical contribuyen a la distancia, produciendo la distancia de Manhattan.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-ep0409-distancia.html

Figura 4.38: Simulador EP04_09: Transformada de Distância (Camadas de Erosión)

```
%%writefile EP04_09.py
# Código Python
```

```
Overwriting EP04_09.py
```

```
TestSuite("EP04_09.py").run()
```

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

4.20.10 EP04_10 🪙 Separación de *Blobs*, Etiquetado y Descriptores

En una **línea de producción de monedas**, es común que las piezas se toquen entre sí en la cinta transportadora, formando una única mancha conectada en la imagen — un conteo ingenuo erraría el total. La solución clásica combina operaciones morfológicas y análisis de conectividad: primero una **erosión** reduce o rompe conexiones frágiles entre objetos, y luego el **etiquetado de componentes conectados** separa cada objeto en una región distinta. Finalmente, **descriptores geométricos** (área y caja delimitadora) resumen cada componente encontrado.

Ver en Figura 4.39 una simulación de este EP.

4.20.10.1 📋 Directrices de Implementación

1. **Dimensiones de la imagen:** leer los enteros L (filas) y C (columnas) de f.
2. **Dimensiones de B:** leer los enteros L_B (filas) y C_B (columnas) del elemento estructurante.
3. **Elemento estructurante:** leer la matriz B, que contiene valores 0 o 1, fila a fila.
4. **Datos:** leer la matriz binaria f (valores 0 o 1), fila a fila.

5. **Separación:** calcular
$$f_{ero} = f \ominus B$$
usando erosión binaria plana (como en el EP04_04), eliminando conexiones frágiles entre objetos.

6. **Etiquetado:** sobre f_{ero}, identificar componentes conectados usando conectividad definida por la vecindad B. El etiquetado debe seguir un barrido *raster*: al encontrar un píxel 1 aún no etiquetado, asignar una nueva etiqueta entera creciente a partir de 1 y propagar esa etiqueta a toda la región conectada.

7. **Descriptores:** para cada etiqueta k, calcular:
   - **Área:** número de píxeles pertenecientes a la etiqueta;
   - **Caja delimitadora:**
$$(y_{min}, x_{min}, y_{max}, x_{max})$$

8. **Salida:** mostrar el número total de etiquetas y, a continuación, una línea por etiqueta en el formato:
$$k,\ \text{área},\ y_{min},\ x_{min},\ y_{max},\ x_{max}$$

4.20.10.2 📌 Restricciones Computacionales

- La erosión debe aplicarse antes del etiquetado.
- La conectividad es fija y está definida por la vecindad anterior.
- El elemento estructurante B no interfiere en la conectividad del etiquetado.
- Sin *padding* en ninguna etapa.
- El orden de las etiquetas sigue la primera detección en el barrido *raster*.

4.20.10.3 🧠 Fundamentación Teórica

Concepto	Significado	Impacto
Puente fino	Conexión estrecha entre objetos	Puede ser eliminado por la erosión morfológica
Conectividad	Definida por el conjunto $\mathcal{N}(y, x)$	Determina qué píxeles pertenecen al mismo componente
Área	Número de píxeles por componente	Estimación directa del tamaño del objeto
Caja delimitadora	Extensión espacial de la etiqueta	Resumen geométrico del componente

4.20.10.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: entero L
- Línea 2: entero C
- Línea 3: entero L_B
- Línea 4: entero C_B
- Siguientes L_B líneas: matriz B
- Siguientes L líneas: matriz f

Salida:

- Línea 1: número total de etiquetas encontradas
- Líneas siguientes:
$$k,\ \text{área},\ y_{min},\ x_{min},\ y_{max},\ x_{max}$$

```
%%writefile EP04_10.py
# Código Python
```

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap04/sim-ep0410-rotulacao.html

Figura 4.39: Simulador EP04_10: Separación de Blobs, Etiquetado y Descriptores

```
Overwriting EP04_10.py
```

```
TestSuite("EP04_10.py").run()
```

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

Capítulo 5

Transformadas y Compresión

Executar Colab Abrir GitHub

En los capítulos anteriores, todas las operaciones se realizaron **en el dominio espacial**, en el que los algoritmos actúan directamente sobre los valores de intensidad de los píxeles.

En este capítulo se presentará un enfoque complementario: el **dominio de la frecuencia**, en el cual la imagen se representa mediante las variaciones espaciales de intensidad, y no solo por los valores individuales de los píxeles.

El concepto de **frecuencia espacial** describe la rapidez con la que la intensidad varía a lo largo de la imagen. Las variaciones lentas corresponden a **bajas frecuencias**, mientras que los bordes, los detalles finos y los ruidos corresponden a **altas frecuencias**.

Esta representación se basa en el hecho de que cualquier imagen digital discreta puede descomponerse en una combinación de **funciones ortogonales**. La **Transformada de Fourier** utiliza una **base de exponenciales complejas bidimensionales** (equivalentes a senoides con orientación y frecuencia específicas). Otras transformadas, como la **Transformada de Cosenos (DCT)** y la **Transformada *Wavelet* (DWT)**, utilizan diferentes familias de funciones de base — cosenos bidimensionales en el caso de la DCT, y funciones con soporte compacto en el caso de las *wavelets*.

Entre las principales aplicaciones de esta representación se destacan:

1. **Filtrado en el dominio de la frecuencia**, para atenuar o realzar determinadas bandas de frecuencia;
2. **Análisis multirresolución mediante transformadas *wavelet***, que representa estructuras en diferentes escalas;
3. **Compresión de imágenes**, mediante la reducción del número de coeficientes necesarios para representar la imagen.

5.1 Objetivos

Al concluir este capítulo, usted será capaz de:

- **Interpretar el espectro de Fourier** de una imagen, distinguiendo magnitud, fase y componentes de frecuencia;
- **Aplicar el Teorema de la Convolución** para realizar filtrado en el dominio de la frecuencia utilizando la Transformada Rápida de Fourier (FFT);
- **Diseñar y analizar filtros en el dominio de la frecuencia**, comprendiendo el funcionamiento de filtros pasa-baja, pasa-alta y *notch*;
- **Comprender el análisis multirresolución mediante transformadas *wavelet*** y su aplicación en la representación jerárquica de imágenes;
- **Describir el proceso de compresión de imágenes**, incluyendo la Transformada Discreta del Coseno (DCT) y la cuantización de los coeficientes;
- **Seleccionar formatos de almacenamiento de imágenes**, como JPEG, PNG y WebP, de acuerdo con los requisitos de la aplicación.

5.2 Configuración del Entorno

```
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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0
```

5.3 Transformada de Fourier Discreta 2D

El análisis de Fourier se basa en el principio de que cualquier señal periódica puede representarse como una suma de funciones senoidales con diferentes frecuencias, amplitudes y fases. Este concepto también se aplica a las imágenes digitales, permitiendo representarlas en el **dominio de la frecuencia** en lugar del dominio espacial.

La Figura 5.1 ilustra esta descomposición para una señal unidimensional. En el caso de una imagen, la Transformada Discreta de Fourier (DFT) convierte la matriz de intensidades $f(x, y)$ en un conjunto de coeficientes que describe la contribución de las diferentes frecuencias espaciales presentes en la imagen.

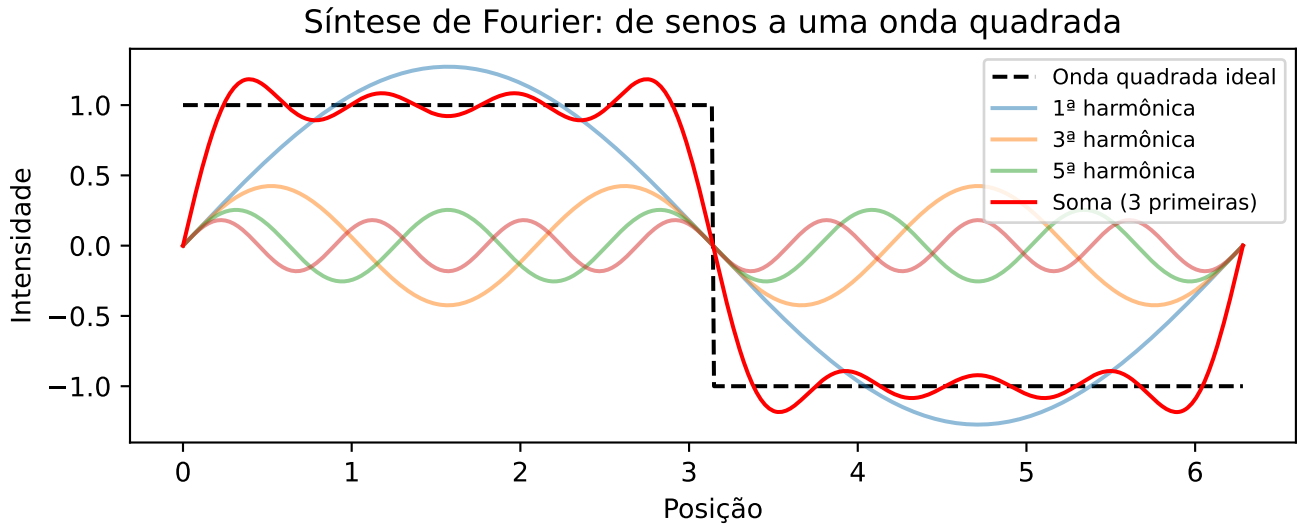


Figura 5.1: Descomposición de Fourier 1D: una onda cuadrada (línea discontinua) se aproxima mediante la suma de las primeras sinusoides (líneas de colores). Cuantos más términos, mejor la aproximación.

5.3.1 Simulador: Reconstruyendo Señales con Senoides

Antes de estudiar imágenes bidimensionales, el simulador de la Figura 5.2 ilustra el principio del análisis de Fourier para señales unidimensionales: **una forma de onda puede aproximarse mediante la suma de senoides con diferentes frecuencias y amplitudes**.

A medida que se añaden nuevos términos, la suma de las senoides (curva negra) se aproxima a la forma de onda de referencia (trazada). El gráfico inferior presenta el espectro de amplitudes, indicando la contribución de cada frecuencia a la reconstrucción de la señal.

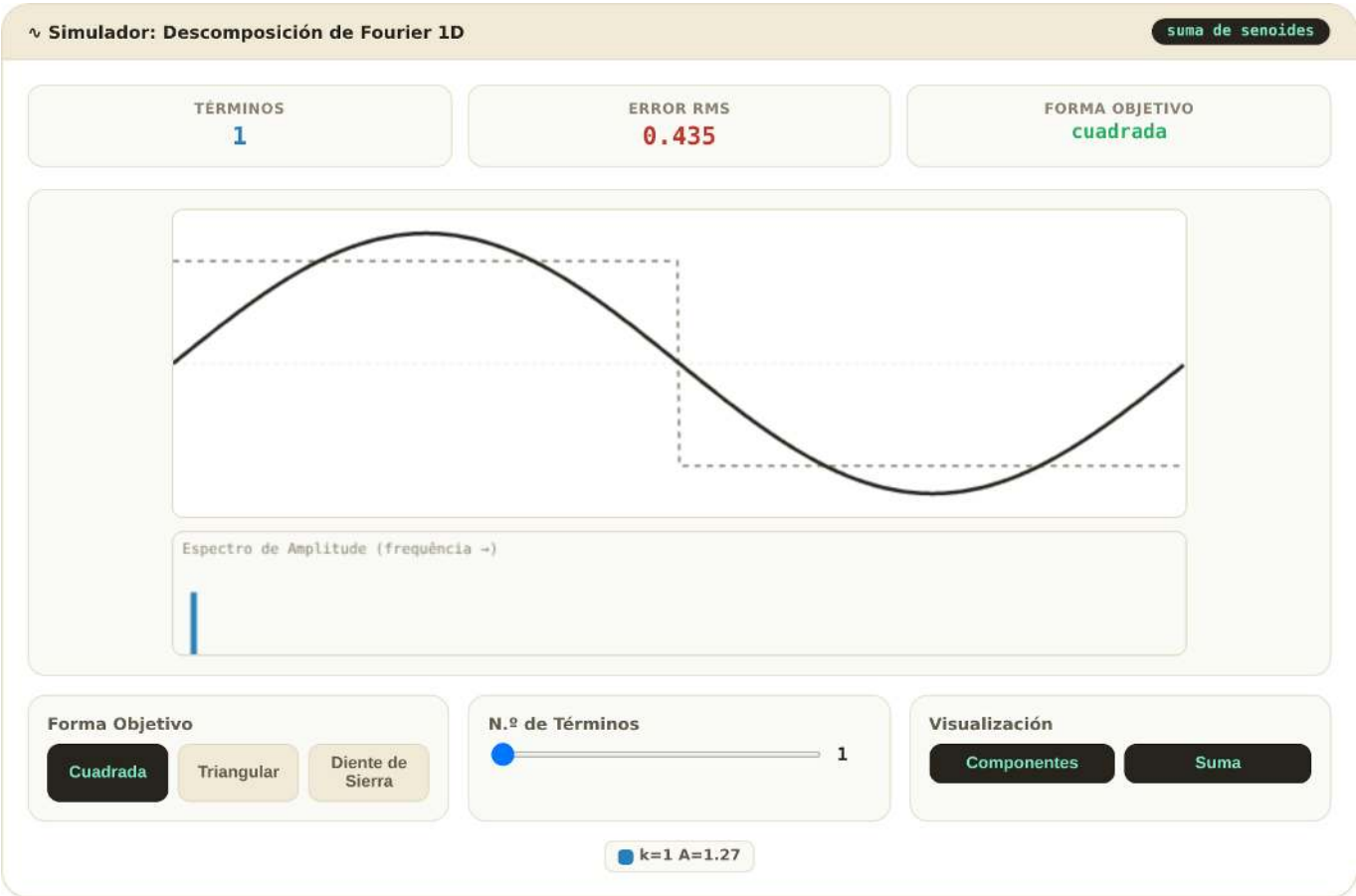
Actividad

Explore el simulador y responda:

1. ¿Cuántos términos son necesarios para obtener una buena aproximación de la onda cuadrada?
2. ¿Cuál de las tres formas de onda converge más rápidamente? Justifique su respuesta.
3. ¿Cómo se altera el espectro de amplitudes al cambiar la onda cuadrada por la triangular?

Respuestas

1. ¿Cuántos términos son necesarios para una buena aproximación de la onda cuadrada?
Con aproximadamente 15 a 20 términos, la forma de la onda ya se aproxima bien a la referencia. Sin embargo, cerca de las discontinuidades permanece una pequeña oscilación, conocida como **fenómeno de Gibbs**, que no desaparece incluso con la adición de más términos.
2. ¿Cuál forma converge más rápidamente? ¿Por qué?
La **onda triangular** converge más rápidamente, pues las amplitudes de sus armónicos decaen más rápido que las de la onda cuadrada y la onda de diente de sierra. Como consecuencia, pocos términos ya producen una buena aproximación.
3. ¿Cómo cambia el espectro entre la onda cuadrada y la triangular?
Ambas poseen únicamente **armónicos impares**, pero, en la onda triangular, las amplitudes disminuyen mucho más rápidamente. Así, pocos armónicos son suficientes para reconstruir la señal con buena precisión.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap05/sim-05-freq.html

Figura 5.2: Simulador interactivo de la descomposición de Fourier 1D: visualización de la suma de senoides con diferentes frecuencias, amplitudes y fases. Añada términos y observe la convergencia hacia formas de onda arbitrarias.

5.3.2 Interpretación del espectro de frecuencia

Al aplicar la Transformada Discreta de Fourier (DFT) a una imagen y visualizar el módulo de sus coeficientes (ver Figura 5.5), se obtiene el **espectro de magnitud**, que muestra la distribución de las frecuencias espaciales presentes en la imagen.

El coeficiente ubicado en el origen de la DFT, denominado **componente DC** (*Direct Current*), corresponde a la frecuencia nula y representa la intensidad media de la imagen. Por convención, dicho coeficiente se almacena en la esquina superior izquierda del espectro. Para facilitar su interpretación, se aplica la operación **FFT Shift**, que desplaza la componente DC al centro de la imagen. Tras este desplazamiento, las bajas frecuencias se concentran en la región central, mientras que las altas frecuencias se sitúan cerca de los bordes, tal como resume la Tabla 5.1.

Tabla 5.1: Correspondencia entre las regiones del espectro de magnitud tras la aplicación del FFT Shift.

Región del espectro	Componentes predominantes	Ejemplos en la imagen
Centro (bajas frecuencias)	Variaciones espaciales lentas	Iluminación, regiones homogéneas y formas globales
Región intermedia (frecuencias medias)	Variaciones de escala intermedia	Texturas y patrones repetitivos
Bordes (altas frecuencias)	Variaciones espaciales rápidas	Contornos, detalles finos y ruido

Esta organización facilita la interpretación del espectro y el diseño de filtros. La atenuación de las bajas frecuencias reduce las variaciones globales de intensidad, mientras que la atenuación de las altas frecuencias suaviza la imagen al disminuir los detalles finos y parte del ruido.

5.3.3 El Experimento de la Rejilla: Construyendo una Imagen a partir de un Único Coeficiente

Antes de presentar la formulación matemática de la Transformada Discreta de Fourier (DFT), es útil analizar su inversa, denominada Transformada Discreta Inversa de Fourier (IDFT). Considere un espectro en el que todos los coeficientes sean nulos, excepto uno. Un ejemplo de esta construcción se presenta en el código de la Figura 5.3 y puede explorarse interactivamente en el simulador de la Figura 5.4..

La imagen reconstruida es una senoide bidimensional. La posición del coeficiente en el espectro determina su **orientación** y su **frecuencia espacial**, mientras que su magnitud y su fase definen, respectivamente, su amplitud y su desplazamiento espacial. Así, cada coeficiente de la DFT representa una componente senoidal, y la imagen original puede reconstruirse mediante la suma de todas estas componentes.

```
import cv2

N_grid = 100
espectro_vazio = np.zeros((N_grid, N_grid), dtype=complex)

# Encendiendo un unico punto (frecuencia) fuera del centro
u0, v0 = 10, 5
espectro_vazio[N_grid//2 - v0, N_grid//2 - u0] = 1000

# Regresando al dominio espacial (IDFT)
onda_2d = np.real(np.fft.ifft2(np.fft.ifftshift(espectro_vazio)))

onda_vis = cv2.normalize(onda_2d, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)
espectro_vis = cv2.normalize(
    np.abs(espectro_vazio), None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

# Resalte visual del punto
espectro_color = cv2.cvtColor(espectro_vis, cv2.COLOR_GRAY2BGR)
```

```
cv2.circle(espectro_color, (N_grid//2 - u0, N_grid//2 - v0), 2, (0, 0, 255), -1)

mm.show([espectro_color, onda_vis],
        titles=["Espectro (1 punto activo)", "Onda 2D resultante (IDFT)"],
        cols=2, figsize=(10, 4))
```

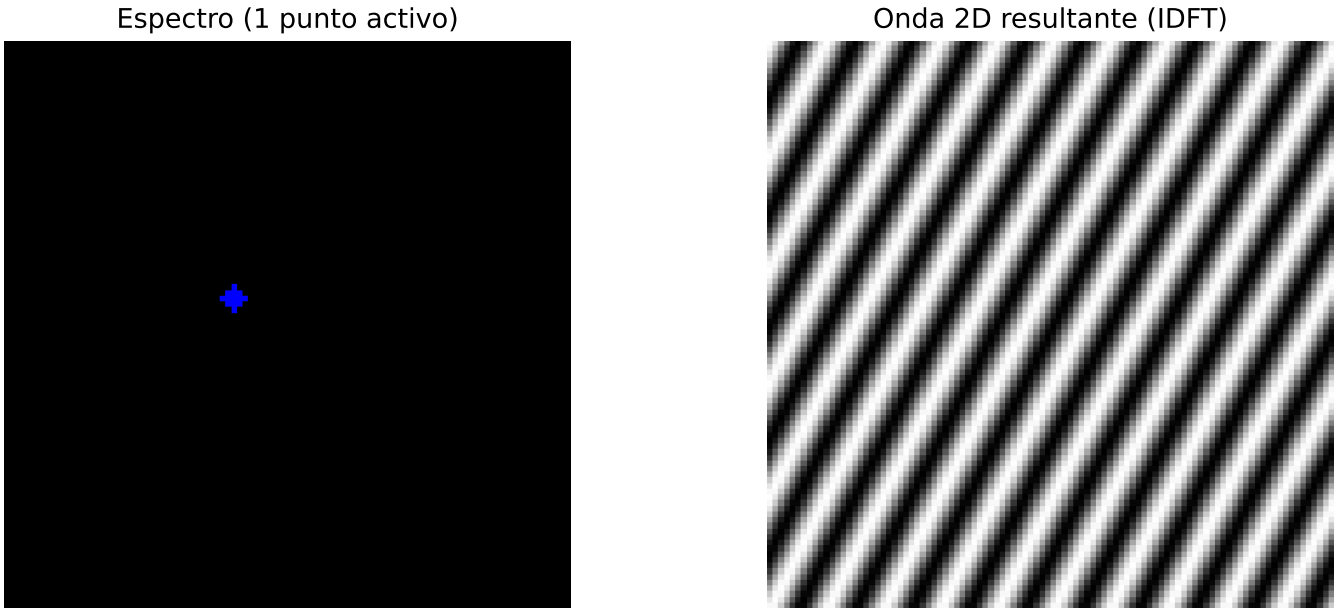
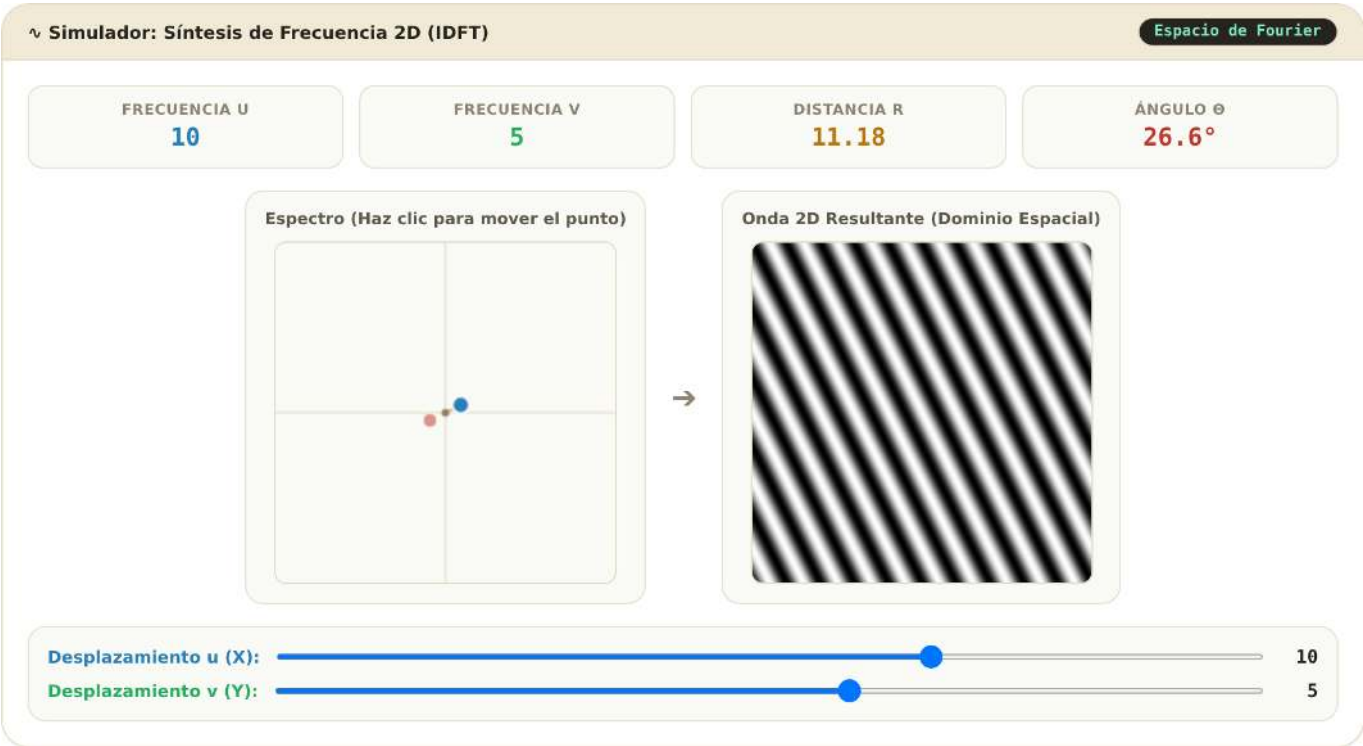


Figura 5.3: Toda frequencia en el espectro (punto aislado) corresponde a una onda sinusoidal 2D rotada en el dominio espacial.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap05/sim-05-grade-2d.html

Figura 5.4: Simulador interactivo de la síntesis de Fourier 2D. Cambie la posición horizontal (u) y vertical (v) del coeficiente en el espectro de frecuencias centrado y observe cómo la distancia respecto al centro determina la frecuencia espacial (grosor) y el ángulo determina la orientación de la onda sinusoidal generada.

5.3.4 Definición Matemática

Considere una imagen $f(x, y)$ con dimensiones $M \times N$. Su **Transformada Discreta de Fourier 2D** (DFT) se define por:

$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y)\, e^{-j2\pi(\frac{ux}{M} + \frac{vy}{N})} \tag{5.1}$$

donde $u = 0, 1, \ldots, M-1$ y $v = 0, 1, \ldots, N-1$ representan las frecuencias discretas en las direcciones horizontal y vertical, respectivamente. El término exponencial corresponde a una senoide bidimensional, cuya frecuencia y orientación están determinadas por los índices (u, v).

La **Transformada Discreta Inversa de Fourier 2D** (IDFT) reconstruye la imagen original a partir de sus coeficientes:

$$f(x, y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v)\, e^{j2\pi(\frac{ux}{M} + \frac{vy}{N})} \tag{5.2}$$

Las Ecuaciones Ecuación 5.1 y Ecuación 5.2 muestran que la DFT y la IDFT forman un par de transformaciones: la primera convierte la imagen al dominio de la frecuencia, mientras que la segunda reconstruye exactamente la imagen original a partir de sus coeficientes.

Sobre el símbolo j

El término j denota la **unidad imaginaria**, definida por $j^2 = -1$. En ingeniería y procesamiento de señales, se adopta j en lugar de i para evitar conflictos con la notación de corriente eléctrica. Su uso en la exponencial compleja, regida por la fórmula de Euler ($e^{j\theta} = \cos\theta + j \sin\theta$), permite representar de forma compacta la amplitud y la fase de cada frecuencia espacial presente en la imagen.

¿Qué es el componente DC?

El coeficiente $F(0, 0)$, denominado **componente DC** (*Direct Current*), es igual a la suma de las intensidades de todos los píxeles de la imagen (ver Figura 5.5):

$$F(0, 0) = MN\, \bar{f},$$

donde $\bar{f}$ es la intensidad media de la imagen. Por ello, el componente DC representa el nivel medio de intensidad y, en la mayoría de las imágenes naturales, posee la mayor magnitud del espectro.
Los demás coeficientes representan variaciones en torno a esa media. Tras la aplicación del **FFT Shift**, el componente DC se desplaza al centro del espectro, concentrando las bajas frecuencias en la región central y las altas frecuencias en los bordes.

5.3.5 Magnitud y Fase

Cada coeficiente de la Transformada Discreta de Fourier (DFT) es un número complejo y puede escribirse como

$$F(u, v) = R(u, v) + j\, I(u, v),$$

donde $R(u, v)$ e $I(u, v)$ corresponden, respectivamente, a las partes **real** e **imaginaria** del coeficiente. De la Ecuación Ecuación 5.1, se obtienen

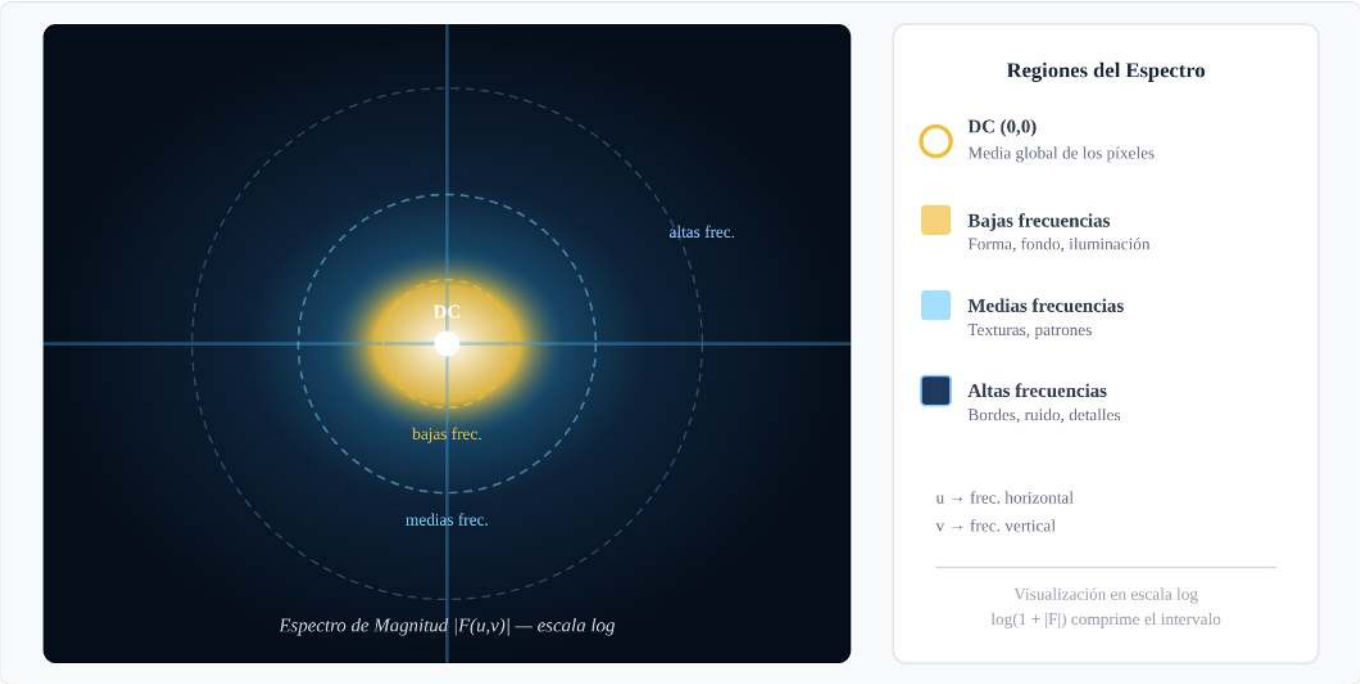


Figura 5.5: Diagrama conceptual del espectro de Fourier 2D centrado.

$$R(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \cos\Big(2\pi \left(\frac{ux}{M} + \frac{vy}{N}\right)\Big),$$

y

$$I(u,v) = -\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \sin\Big(2\pi \left(\frac{ux}{M} + \frac{vy}{N}\right)\Big).$$

A partir de esta representación, se definen dos magnitudes fundamentales:

- **Magnitud**, que indica la intensidad de la componente de frecuencia,

$$|F(u,v)| = \sqrt{R(u,v)^2 + I(u,v)^2};$$

- **Fase**, que determina la alineación (o desplazamiento) espacial de la componente,

$$\phi(u,v) = \operatorname{atan2}(I(u,v),\, R(u,v)).$$

Así, cada coeficiente también puede escribirse en su forma polar,

$$F(u,v) = |F(u,v)|\, e^{j\phi(u,v)}.$$

El espectro de Fourier puede, por tanto, visualizarse mediante dos imágenes distintas: el **espectro de magnitud**, normalmente utilizado para analizar la distribución de las frecuencias, y el **espectro de fase**, que describe la organización espacial de las componentes senoidales.

Aunque el espectro de magnitud sea el más utilizado para la inspección visual, la fase contiene gran parte de la información estructural de la imagen. La combinación de magnitud y fase permite reconstruir exactamente la imagen original mediante la IDFT.

5.3.6 ¿Qué transportan la magnitud y la fase?

Una demostración clásica consiste en combinar la magnitud de una imagen con la fase de otra y reconstruir el resultado. Este experimento evidencia que:

- **La fase** preserva la estructura espacial de la imagen, incluida la posición de los objetos, sus contornos y su geometría. Pequeñas alteraciones en la fase pueden provocar grandes cambios visuales.
- **La magnitud** controla cómo se distribuye la energía entre las frecuencias espaciales, influyendo principalmente en el contraste y la textura.

Cuando una imagen se reconstruye con la magnitud de A y la fase de B, el resultado tiende a **parecerse más a B que a A**, evidenciando que la fase es el principal componente responsable de la organización espacial de la escena. Sin embargo, la magnitud sigue siendo importante, ya que modula el contraste de las estructuras reconstruidas. Así, una reconstrucción fiel depende de la combinación coherente entre magnitud y fase.

Un ejemplo de este comportamiento se presenta en la Figura 5.6.

Analogía con el Audio: Limitaciones y Precauciones

La fase de una señal desempeña papeles distintos en audio e imágenes:

- **Audio estéreo o multicanal:** la fase relativa entre los canales es fundamental para la percepción de la posición de las fuentes sonoras, mediante las diferencias interaurales de tiempo (ITD, *Interaural Time Differences*).
- **Audio monoaural:** la fase absoluta ejerce poca influencia perceptual directa.
- **Imágenes (DFT):** la fase es el principal factor responsable de la organización espacial de la escena, mientras que la magnitud modula el contraste y la distribución de la energía entre las frecuencias.

En ambos dominios, la **magnitud** está relacionada con la intensidad de los componentes de frecuencia: en audio, influye en el timbre y la intensidad percibida; en imágenes, influye en el contraste y la textura.

```
#   Experimento: La Importancia de la Fase
#   Carga de la imagen
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))

# Crear una imagen B sintética (patrón geométrico)
img_b = np.zeros((400, 400), dtype=np.uint8)
cv2.rectangle(img_b, (100, 100), (300, 300), 255, -1)
cv2.circle(img_b, (200, 200), 150, 128, 10)

FA = np.fft.fft2(img_a)
FB = np.fft.fft2(img_b)

# Intercambio de Fase
rec_A_mag_B_fase = np.real(np.fft.ifft2(np.abs(FA) * np.exp(1j * np.angle(FB))))
rec_B_mag_A_fase = np.real(np.fft.ifft2(np.abs(FB) * np.exp(1j * np.angle(FA))))
```

```
mm.show(
    [img_a, img_b, rec_A_mag_B_fase, rec_B_mag_A_fase],
    titles=["Imagen A", "Imagen B", "Mag(A) + Fase(B)", "Mag(B) + Fase(A)"],
    cols=4, figsize=(16, 4)
)

print("  La fase preserva bordes y contornos; la magnitud controla contraste y")
print("textura. En audio estéreo, la fase afecta la localización espacial; en")
print("imágenes, determina la organización de la escena.")
```

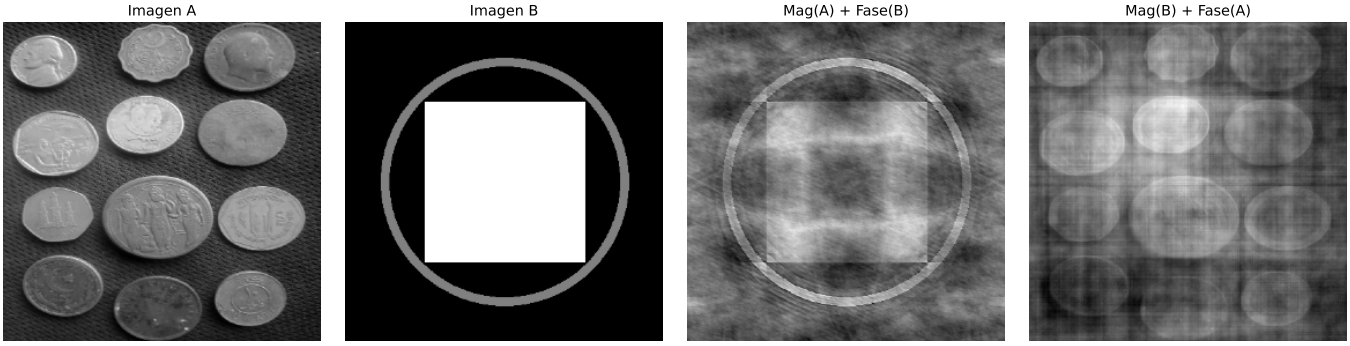


Figura 5.6: Experimento de intercambio de fase: Imagen A (monedas) e Imagen B (patrón geométrico) reconstruidas con magnitudes y fases intercambiadas. El resultado muestra que la estructura visual es **mucho más sensible a la fase** que a la magnitud: cuando la fase de B se mantiene, la imagen resultante preserva la organización espacial de B, incluso con la magnitud de A. La magnitud, por su parte, influye principalmente en el contraste y la textura. Observe que la calidad de la reconstrucción no es perfecta — hay artefactos visibles —, evidenciando la interdependencia entre fase y magnitud para una representación fiel de la imagen.

```
 La fase preserva bordes y contornos; la magnitud controla contraste y
textura. En audio estéreo, la fase afecta la localización espacial; en
imágenes, determina la organización de la escena.
```

5.4 Teorema de la Convolución y Estrategias de Filtrado

El **Teorema de la Convolución** establece una relación fundamental entre los dominios espacial y de la frecuencia:

$$f(x,y) \circledast h(x,y) \overset{\mathcal{F}}{\longleftrightarrow} F(u,v)\,H(u,v) \tag{5.3}$$

donde $\circledast$ representa la **convolución circular discreta**. Así, la convolución entre una imagen $f(x,y)$ y un filtro $h(x,y)$ puede sustituirse por la multiplicación de sus espectros.

En la práctica, para obtener el mismo resultado de la convolución lineal realizada en el dominio espacial, se aplica **relleno con ceros** (*zero-padding*) antes de la Transformada Rápida de Fourier (FFT), evitando artefactos en los bordes de la imagen.

Sin embargo, no siempre el filtrado en el dominio de la frecuencia es la alternativa más eficiente. Para filtros como el gaussiano y el filtro de la media (*Box Filter*), la propiedad de **separabilidad** permite reducir significativamente el costo computacional de la convolución en el dominio espacial.

5.4.1 *Kernel* Separable vs. No separable

Un ***kernel* separable** puede escribirse como el producto externo de dos vectores unidimensionales,

$$H = v\,h^T,$$

lo que permite que la convolución bidimensional se reemplace por dos convoluciones unidimensionales consecutivas: una en la dirección horizontal y otra en la vertical.

En cambio, un ***kernel* no separable** no admite esta descomposición y, por lo tanto, su convolución debe realizarse directamente sobre la vecindad bidimensional.

En la práctica, para un *kernel* de dimensión $K \times K$, la convolución directa requiere K^2 multiplicaciones por píxel, mientras que un *kernel* separable requiere solo $2K$ multiplicaciones, reduciendo significativamente el costo computacional.

5.4.2 Análisis de Eficiencia Computacional

Considere una imagen de dimensiones $M \times N$ y un filtro cuadrado de tamaño $K \times K$. La Tabla 5.2 compara la complejidad de las principales estrategias de filtrado.

Tabla 5.2: Comparación de la complejidad de la convolución directa, separable y vía Transformada Rápida de Fourier (FFT).

Método de Filtrado	Complejidad Asintótica	Dependencia de K	Aplicación típica
Espacial no separable	$\mathcal{O}(MNK^2)$	Cuadrática	*Kernels* pequeños y no separables
Espacial separable	$\mathcal{O}(MNK)$	Lineal	Filtros Gaussiano y de la media
Vía FFT	$\mathcal{O}(MN\log(MN))$	Independiente de K	*Kernels* grandes

Para *kernels* pequeños, la convolución espacial, especialmente cuando el filtro es separable, suele ser más eficiente debido al bajo costo de las operaciones. A medida que el tamaño del *kernel* aumenta, el filtrado vía FFT se vuelve más ventajoso, ya que su costo prácticamente no depende de la dimensión del filtro.

5.4.3 Discusión de los resultados experimentales

El gráfico obtenido en el ensayo con la imagen de las monedas (2560×1920), presentado en la Figura 5.7, confirma el comportamiento previsto por el análisis de complejidad computacional.

1. **Convolución no separable ($\mathcal{O}(MNK^2)$)**
   La convolución directa presenta un crecimiento cuadrático con el tamaño del *kernel*. Para valores pequeños de K, el costo es bajo, pero aumenta rápidamente a medida que el *kernel* crece, volviéndose inviable para aplicaciones en tiempo real.

2. **Filtrado mediante FFT ($\mathcal{O}(MN\log(MN))$)**
   El costo de la FFT depende únicamente del tamaño de la imagen, siendo independiente de K. Por ello, su rendimiento permanece aproximadamente constante al variar el *kernel*, lo que la hace ventajosa para filtros grandes o no separables.

3. **Convolución separable ($\mathcal{O}(MNK)$)**
   La descomposición del *kernel* en dos filtros unidimensionales reduce significativamente el costo computacional. En la práctica, este enfoque tiende a ser el más eficiente para filtros separables, especialmente en implementaciones optimizadas.

En general, la elección del método depende del tamaño y la estructura del *kernel*. Los filtros separables son más eficientes en el dominio espacial, mientras que la FFT resulta más ventajosa para *kernels* grandes o múltiples convoluciones en el dominio de la frecuencia.

$$g = \mathcal{F}^{-1}[\mathcal{F}(f)\cdot\mathcal{F}(h)] \quad \text{(FFT)} \qquad g = f \circledast h \quad \text{(convolución directa)} \qquad g = (f \circledast v) \circledast h^T \quad \text{(separable)} \tag{5.4}$$

donde:

- $f(x, y)$ representa la imagen de entrada;
- $h(x, y)$ es el *kernel* bidimensional del filtro;
- v y h^T son, respectivamente, los vectores vertical y horizontal que componen el *kernel* separable.

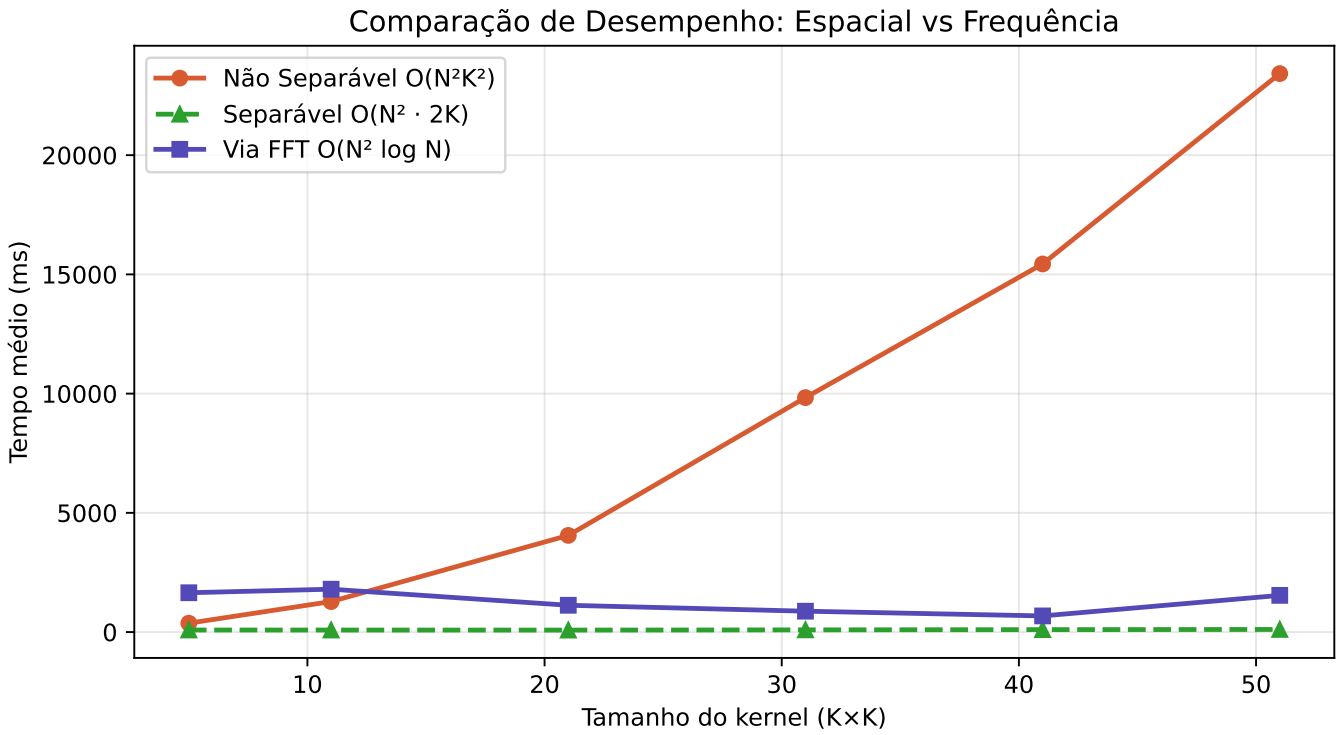


Figura 5.7: Comparación de eficiencia: Convolución No Separable (Espacial 2D), Separable (Espacial 1D) y vía FFT.

! El problema de la convolución circular (*wrap-around*)

La Transformada Discreta de Fourier (DFT) asume que la imagen se **extiende periódicamente en el espacio**, es decir, que sus bordes se repiten indefinidamente.
En esta condición, la multiplicación en el dominio de la frecuencia corresponde a una **convolución circular** en el dominio espacial. Como consecuencia, regiones opuestas de la imagen (parte superior e inferior, izquierda y derecha) interactúan artificialmente, tal como se ilustra en Figura 5.8.
La aplicación de *zero-padding* antes de la FFT reduce este efecto al extender la imagen con valores nulos en los bordes, aproximando el resultado a la convolución lineal. Este comportamiento puede interpretarse a la luz del Teorema de la Convolución, presentado en Figura 5.9.

```
#   Definir M y N
M, N = img_gray.shape  # línea agregada

# Simulación de un filtro de desplazamiento brusco
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))

# Filtrado SIN padding (causa el 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", "Filtrado sin Padding (Fuga)"], cols=2, figsize=(10, 4))
```

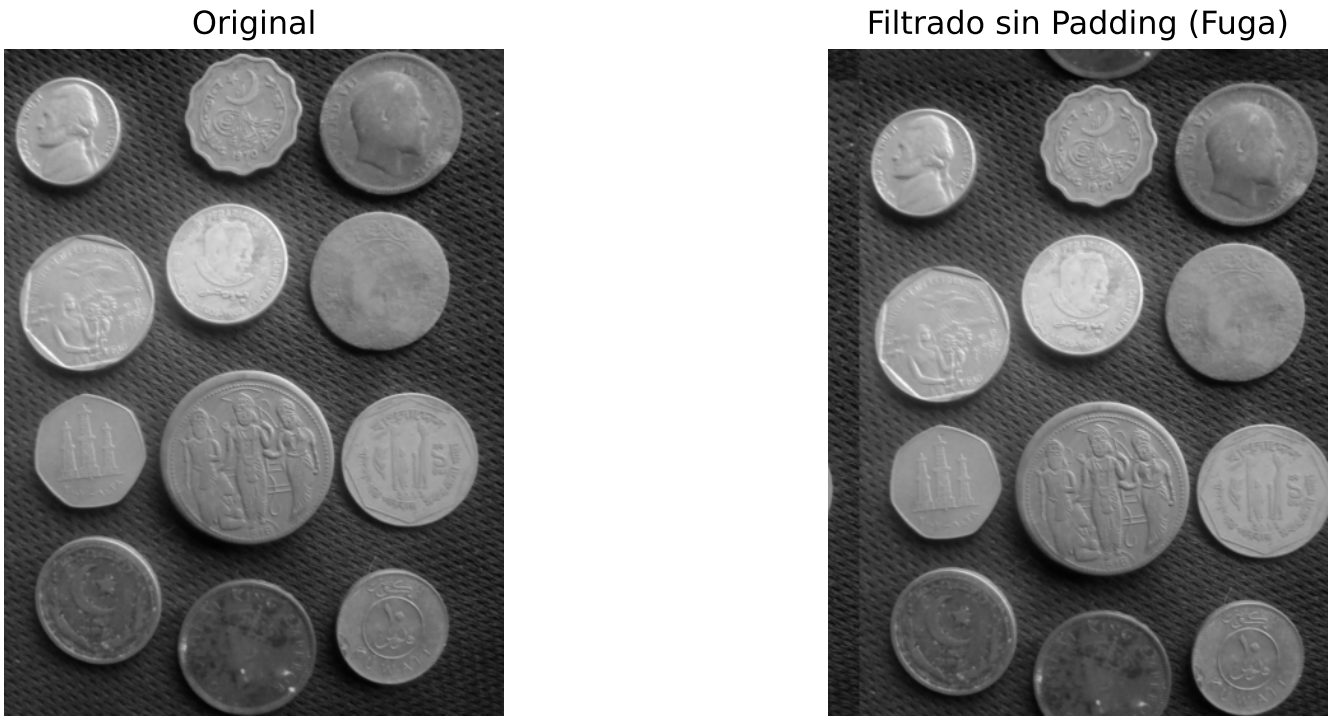


Figura 5.8: Sin *padding*, un desplazamiento severo hace que la imagen se filtre al lado opuesto (convolución circular).

```
#   Kernel Gaussiano 11×11
sigma  = 3.0
K      = 11
ks     = np.arange(K) - K // 2
gauss1d = np.exp(-ks**2 / (2 * sigma**2))
gauss1d /= gauss1d.sum()
kernel  = np.outer(gauss1d, gauss1d)    # kernel 2D separable

#   Método 1: Convolución espacial directa
f_float  = img_gray.astype(np.float64)
conv_esp = cv2.filter2D(f_float, -1, kernel, borderType=cv2.BORDER_CONSTANT)

#   Método 2: Multiplicación en frecuencia (vía FFT)
M, N     = f_float.shape
# Padding para convolución lineal (evita aliasing circular)
Mpad     = 2 ** int(np.ceil(np.log2(M + K - 1)))
Npad     = 2 ** int(np.ceil(np.log2(N + K - 1)))

# Posiciona el kernel con el origen en el (0,0) y padding con ceros
kernel_pad         = np.zeros((Mpad, Npad))
kh, kw             = kernel.shape
kernel_pad[:kh, :kw] = kernel

F_img   = np.fft.fft2(f_float,  (Mpad, Npad))
F_kern  = np.fft.fft2(kernel_pad)
conv_freq = np.real(np.fft.ifft2(F_img * F_kern))

# Recorte para compensar el desplazamiento introducido por el posicionamiento del kernel
```

```
offset   = K // 2
conv_freq_crop = conv_freq[offset:offset+M, offset:offset+N]

#   Verificación numérica
diff = np.abs(conv_esp - conv_freq_crop)
print(f"Diferenza máxima  (|conv_esp - conv_freq|): {diff.max():.2e}")
print(f"Diferenza media   (|conv_esp - conv_freq|): {diff.mean():.2e}")
print(f"→ Teorema de la Convolución verificado numéricamente.")

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",
        "Convolución espacial",
        "Multiplicación en frecuencia",
        f"Diferencia (máx={diff.max():.1e})"
    ],
    cols=4, figsize=(16, 5)
)
```

```
Diferenza máxima  (|conv_esp - conv_freq|): 2.56e-13
Diferenza media   (|conv_esp - conv_freq|): 2.76e-14
→ Teorema de la Convolución verificado numéricamente.
```

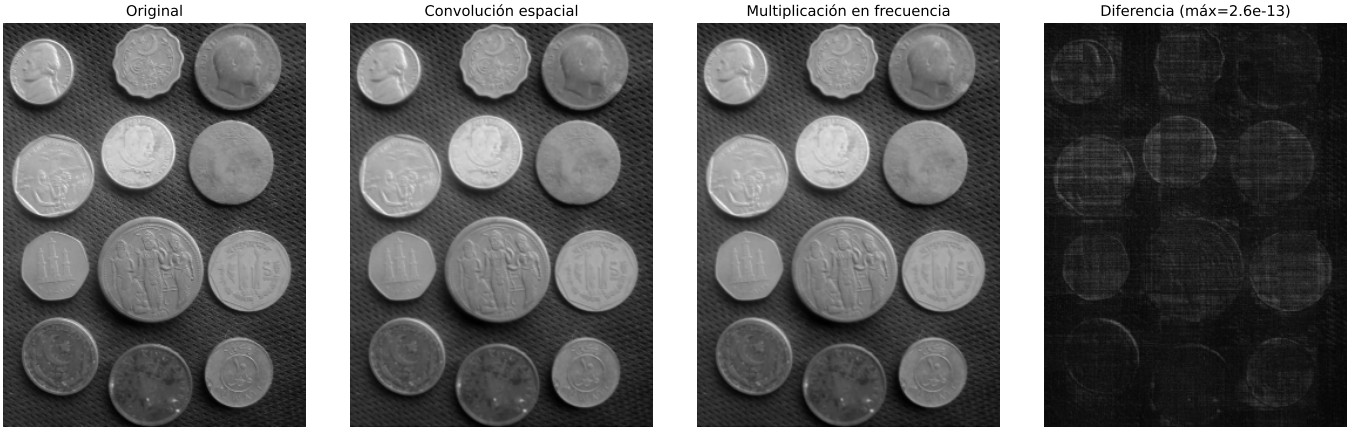


Figura 5.9: Verificación del Teorema de la Convolución: la diferencia píxel a píxel entre la convolución espacial (cv2.filter2D) y la multiplicación en frecuencia (FFT) es numéricamente nula — confirmando la equivalencia teórica.

ℹ Sobre la diferencia numérica

La diferencia residual del orden de 10^{-13} no viola el Teorema de la Convolución, sino que refleja **limitaciones computacionales** inherentes a la aritmética de coma flotante (doble precisión, $\sim 10^{-16}$) y al **orden de las operaciones** entre los dos métodos:

- **Convolución espacial:** suma ponderada de vecinos con redondeos sucesivos.
- **Convolución en frecuencia:** involucra tres transformadas FFT y una multiplicación compleja, sujeta a errores de truncamiento y cuantización.

Por lo tanto, la igualdad teórica es exacta, pero la implementación numérica produce una diferencia prácticamente nula (error relativo $< 10^{-12}$), lo que confirma el teorema dentro de la precisión de la

máquina.

5.5 Filtros en el dominio de la frecuencia

Un filtro en el dominio de la frecuencia puede interpretarse como una **función de transferencia aplicada al espectro de la imagen**. En esta representación, cada coeficiente de frecuencia se multiplica por un valor entre 0 y 1, que determina su atenuación o preservación. La forma de esta función define el efecto visual del filtro.

Corte abrupto y *ringing*. Los filtros ideales con transición instantánea en una frecuencia de corte D_0 producen discontinuidades en el dominio de la frecuencia. Esta discontinuidad se refleja en el dominio espacial como oscilaciones cercanas a los bordes, conocidas como *ringing*. Este efecto está asociado a la convolución con funciones de soporte infinito en el espacio, como la función *sinc*, tal como se ilustra en la Figura 5.10.

Filtros con transición suave. Alternativas como los filtros gaussiano y Butterworth suavizan la transición entre las regiones de paso y de rechazo, reduciendo el *ringing*. En contrapartida, esta suavización implica una frontera de separación menos definida entre las frecuencias preservadas y las atenuadas.

```
# Simulando el Filtro Ideal en la Frecuencia (Cilindro) y su representación Espacial (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)

# Frecuencia: Cilindro Ideal (1 en el centro, 0 fuera del radio 20)
H_freq = np.zeros((N_grid, N_grid))
H_freq[D <= 20] = 1

# Espacio: La inversa resulta en la famosa Sinc 2D
h_space = np.fft.fftshift(np.real(np.fft.ifft2(np.fft.ifftshift(H_freq))))

fig, ax = plt.subplots(1, 2, subplot_kw={'projection': '3d'}, figsize=(12, 4))
ax[0].plot_surface(U, V, H_freq, cmap='viridis', edgecolor='none')
ax[0].set_title("Frequência: Filtro Ideal (Cilindro)")
ax[0].set_zlim(0, 1.2)

ax[1].plot_surface(U, V, h_space, cmap='plasma', edgecolor='none')
ax[1].set_title("Espaço Real: Ondulações da Sinc (Causa do Ringing)")
plt.tight_layout(); plt.show()
```

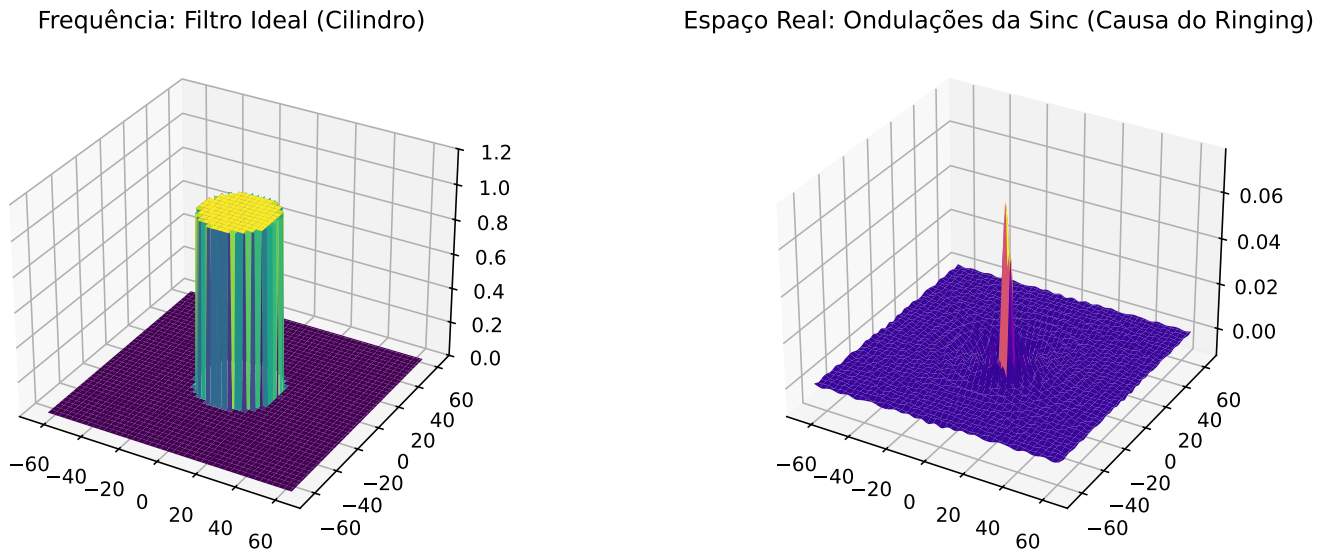


Figura 5.10: Una Dualidad Peligrosa: El corte abrupto en la Frecuencia (Cilindro) se transforma obligatoriamente en una Sinc espacial. Sus ondulaciones causan el *ringing* fantasma en los bordes de la imagen.

5.5.1 Filtros Pasa-Bajos

Los filtros pasa-bajos atenúan los componentes de alta frecuencia, lo que resulta en un suavizado de la imagen y una reducción del ruido. Tras la centralización del espectro (FFT Shift), la distancia de cada punto al centro viene dada por:

$$D(u,v) = \sqrt{\left(u - \tfrac{M}{2}\right)^2 + \left(v - \tfrac{N}{2}\right)^2} \tag{5.5}$$

Filtro Ideal (LPFI):

$$H_{\text{ideal}}(u,v) = \begin{cases} 1, & D(u,v) \leq D_0 \\ 0, & D(u,v) > D_0 \end{cases} \tag{5.6}$$

El corte abrupto en D_0 introduce discontinuidades en el dominio de la frecuencia, lo que da lugar a oscilaciones en el dominio espacial conocidas como *ringing*. Este efecto está asociado con la convolución con funciones de soporte infinito.

Filtro Gaussiano (LPFG):

$$H_{\text{gauss}}(u,v) = e^{-D^2(u,v)/(2\sigma^2)} \tag{5.7}$$

La suavidad de la función gaussiana en el dominio de la frecuencia evita discontinuidades, lo que elimina el *ringing* y produce una transición gradual entre las frecuencias preservadas y las atenuadas.

Filtro de Butterworth (LPFB) de orden n:

$$H_{\text{BW}}(u,v) = \frac{1}{1 + [D(u,v)/D_0]^{2n}} \tag{5.8}$$

El parámetro n controla la suavidad de la transición entre el paso y el rechazo de frecuencias. Los valores pequeños producen transiciones suaves, mientras que los valores grandes aproximan el comportamiento del filtro ideal, con un mayor riesgo de *ringing*. En la Figura 5.11. se presenta un ejemplo comparativo.

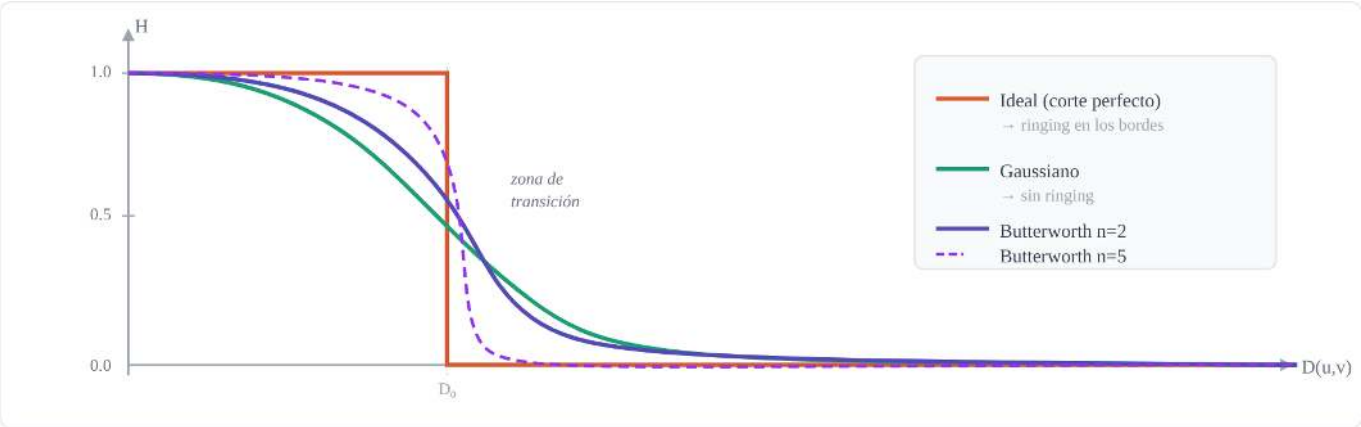


Figura 5.11: Filtros pasa-baja.

5.5.2 Filtros Pasa-Alto y Pasa-Banda

Los filtros pasa-alto pueden obtenerse a partir de un filtro pasa-bajo complementario, definido como:

$$H_{\mathrm{HP}}(u,v) = 1 - H_{\mathrm{LP}}(u,v)$$

Este tipo de filtro preserva componentes de alta frecuencia, resaltando bordes y detalles, mientras atenúa regiones de variación suave.

Los filtros pasa-banda preservan solo un rango intermedio de frecuencias, limitado por dos radios D_L y D_H:

$$H_{\mathrm{BP}}(u,v) = H_{\mathrm{LP}}^{(D_H)}(u,v) \cdot \left[1 - H_{\mathrm{LP}}^{(D_L)}(u,v)\right]$$

Este tipo de filtrado es útil cuando se desea eliminar simultáneamente componentes de baja y alta frecuencia, preservando únicamente estructuras de escala intermedia.

Una aplicación importante es la eliminación de **ruido periódico**, en la cual patrones regulares aparecen como picos localizados en el espectro de magnitud. Estos picos pueden atenuarse mediante filtros *notch* (rechaza-banda), posicionados específicamente en las frecuencias no deseadas.

Ejemplos de filtros en el dominio de la frecuencia se presentan en el simulador de la Figura 5.12, Figura 5.13 y Figura 5.14..

```
import io

def distancia_centro(M, N):
    """Matriz de distancias al centro del espectro."""
    u = np.arange(M) - M // 2
    v = np.arange(N) - N // 2
    V, U = np.meshgrid(v, u)
    return np.sqrt(U**2 + V**2)

def aplicar_filtro_freq(img, H):
    """Aplica filtro H (centrado) a imagen mediante 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
```

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap05/sim-05-filtros.html

Figura 5.12: Simulador interactivo de filtros en el dominio de la frecuencia.

```
D    = distancia_centro(M, N)
D0   = 30      # frecuencia de corte
n_bw = 2       # orden del Butterworth

#   Funciones de transferencia
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))

#   Imágenes filtradas
img_ideal = aplicar_filtro_freq(img_gray, H_ideal)
img_gauss = aplicar_filtro_freq(img_gray, H_gauss)
img_bw    = aplicar_filtro_freq(img_gray, H_bw)

#   Perfiles de H(u,v)
def fig2img(fig):
    b = io.BytesIO(); fig.savefig(b, format='png', dpi=100); plt.close(fig); b.seek(0)
    return (plt.imread(b)[:,:,:3]*255).astype(np.uint8)

fig, ax = plt.subplots(figsize=(6, 3))
linha   = M // 2
ax.plot(H_ideal[linha, :], label="Ideal",       color="#D85A30", lw=1.5, ls="--")
ax.plot(H_gauss[linha, :], label="Gaussiano",   color="#1D9E75", lw=1.5)
ax.plot(H_bw[linha, :],    label="Butterworth", color="#534AB7", lw=1.5)
ax.axvline(N//2-D0, color="#aaa", lw=0.8, ls=":")
ax.axvline(N//2+D0, color="#aaa", lw=0.8, ls=":")
ax.set(title="Perfis H(u,v) - linha central", xlabel="v", ylabel="H(u,v)")
ax.legend(fontsize=8); plt.tight_layout()
perfil_img = fig2img(fig)

#   Filtros H visualizados
def H_vis(H):
    return cv2.normalize((H*255).astype(np.uint8), None, 0, 255, cv2.NORM_MINMAX)

mm.show(
    [img_gray, img_ideal, img_gauss, img_bw,
     H_vis(H_ideal), H_vis(H_gauss), H_vis(H_bw), perfil_img],
    titles=[
        "Original", "LPF Ideal", "LPF Gaussiano", "LPF Butterworth (n=2)",
        "H Ideal",  "H Gaussiano","H Butterworth", "Perfiles H(u,v)"
    ],
    cols=4, figsize=(16, 9)
)
```

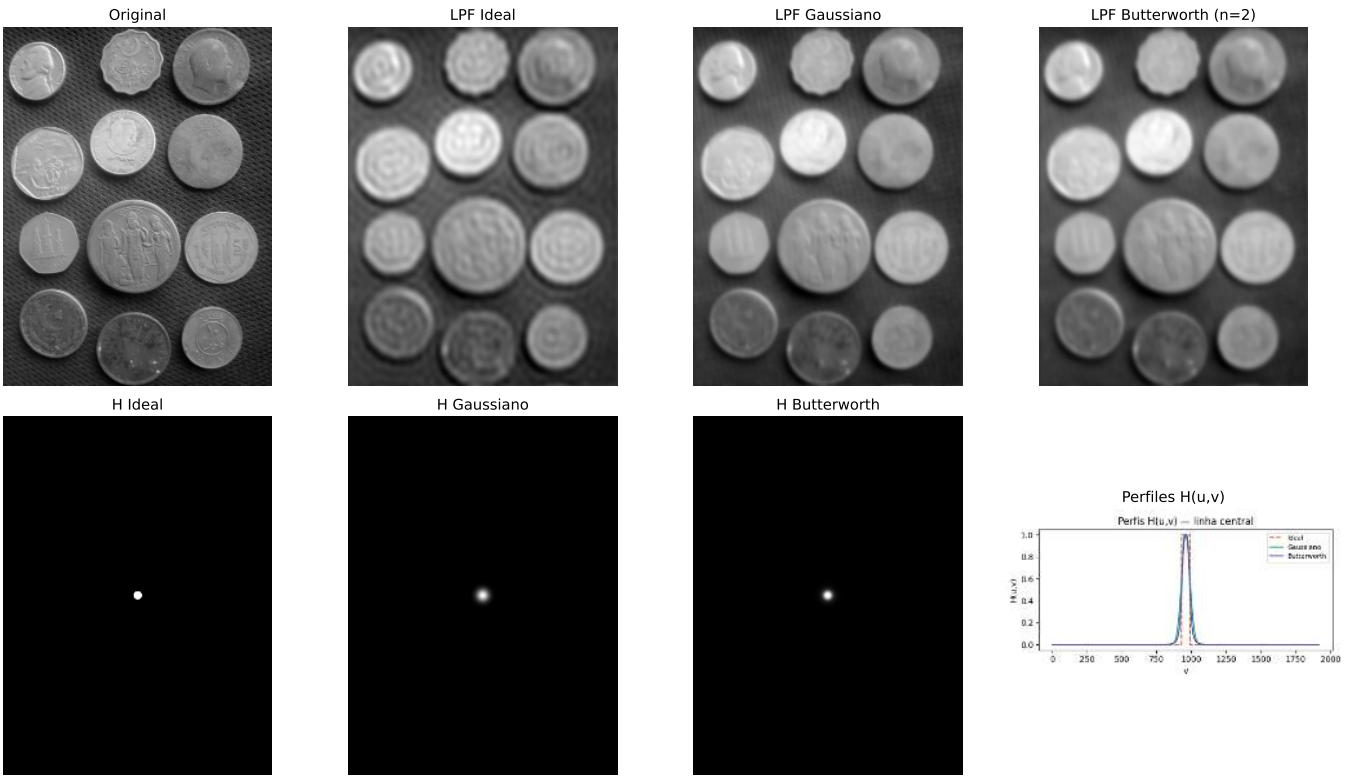


Figura 5.13: Comparación entre filtros pasa-baja: Ideal (D =30), Gaussiano (D =30) y Butterworth (D =30, n=2). Perfiles de H(u,v) a lo largo de una línea central e imágenes filtradas correspondientes.

```
# Filtro pasa-alta: complemento del pasa-baja Gaussiano
# Reutiliza aplicar_filtro_freq() definida en la celda anterior
H_alta   = 1 - H_gauss
img_alta = aplicar_filtro_freq(img_gray, H_alta)

mm.show(
    [img_gray, img_alta],
    titles=["Original", "Pasa-alta Gaussiano ($D_0=30$)"],
    cols=2
)
```

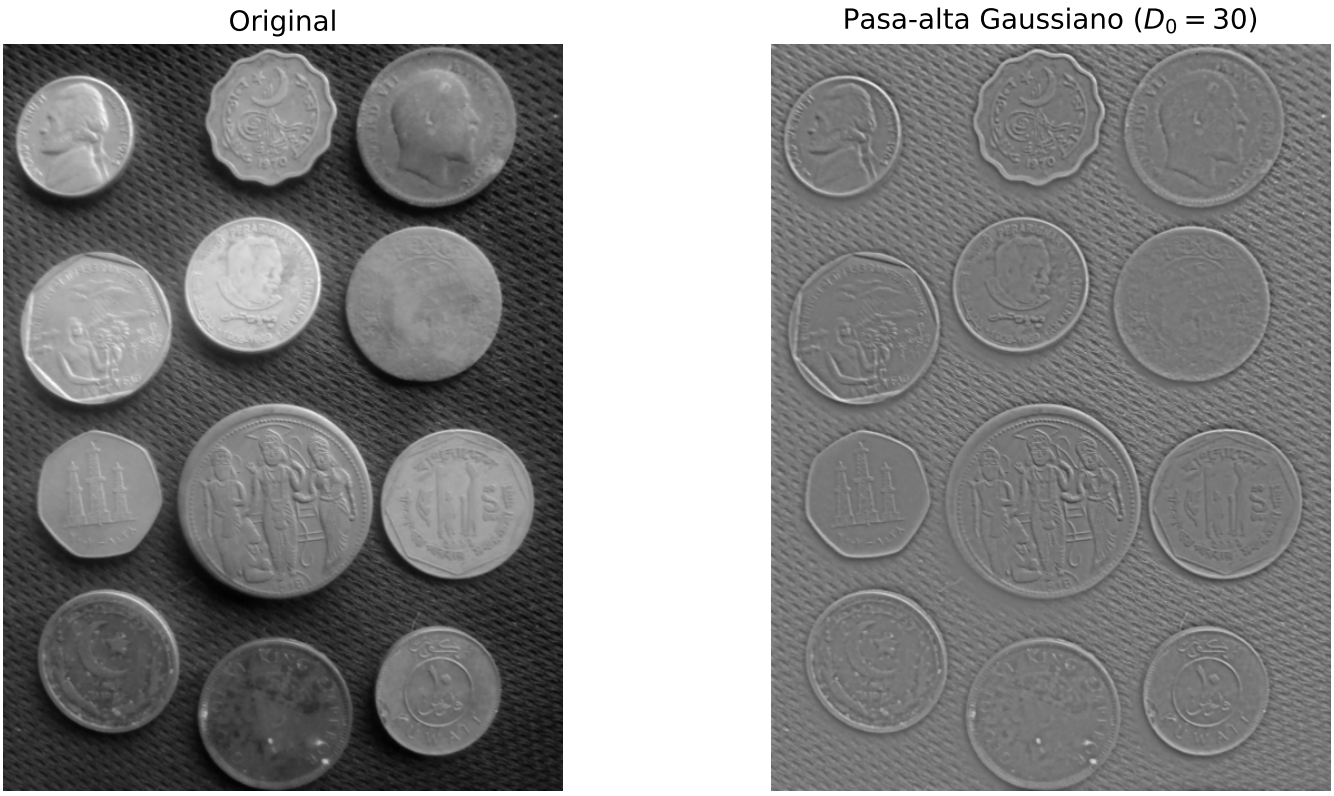


Figura 5.14: Filtro pasa-alta Gaussiano. (a) Original; (b) Filtro pasa-alta (D =30) - los bordes de las monedas y el fondo texturizado se resaltan.

5.5.3 Eliminación de Ruido Periódico

El ruido periódico — asociado a interferencias eléctricas, patrones regulares de sensores o artefactos de digitalización — aparece en el espectro de Fourier como **picos puntuales simétricos alrededor del centro**.

El filtro **rechaza-banda (*notch*)** atenúa selectivamente esas frecuencias, preservando las demás componentes de la imagen. Un ejemplo de aplicación se presenta en la Figura 5.15..

```
#   Imagen con ruido periódico sintético
h_img, w_img = img_gray.shape
x = np.arange(w_img)
y = np.arange(h_img)
X, Y = np.meshgrid(x, y)

# Usar frecuencias enteras y consistentes (importante para restauración perfecta)
u0, v0 = 20, 20  # frecuencias exactas del ruido

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)

#   Espectro de la imagen ruidosa
F_r = np.fft.fftshift(np.fft.fft2(img_ruidosa.astype(np.float64)))
mag_r = np.log1p(np.abs(F_r))
mag_vis = cv2.normalize(mag_r, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

#   Máscara notch
mascara = np.ones((h_img, w_img), dtype=np.float64)
r_notch = 8  # radio del notch (ajuste fino si es necesario)
```

```python
def suprimir_pico(mask, cy, cx, r):
    """Pone a cero un disco de radio r centrado en (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

# Coordenadas centrales
cy, cx = h_img // 2, w_img // 2

# Suprimir los 4 picos simétricos (¡importante!)
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)

#   Filtrado y reconstrucción
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)

# Evaluación
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 restaurada): {psnr:.2f} dB")

#   Visualización
mm.show(
    [img_ruidosa, mag_vis, mascara_vis, img_rest_vis],
    titles=[
        "Con ruido periódico",
        "Espectro (log)",
        "Máscara notch",
        f"Restaurada (PSNR={psnr:.1f} dB)"
    ],
    cols=4,
    figsize=(16, 4)
)
```

```
PSNR (original vs restaurada): 32.93 dB
```

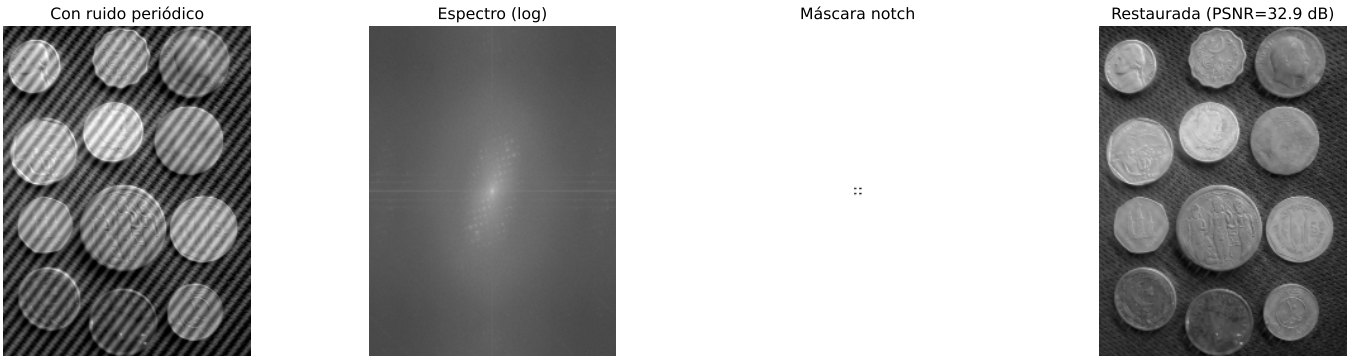


Figura 5.15: Eliminación de ruido periódico mediante filtro *notch* en el dominio de la frecuencia: (a) imagen con ruido senoidal, (b) espectro mostrando los picos del ruido, (c) máscara *notch* centrada en los picos, (d) imagen restaurada.

📌 **Síntesis — Filtros Espectrales**

Filtro	Efecto visual	Artefacto	Uso
Paso-bajo ideal	Suavizado intenso	*Ringing*	Ilustrativo
Paso-bajo gaussiano	Suavizado suave	No presenta *ringing*	Suavizado general
Paso-bajo Butterworth	Suavizado controlado	*Ringing* (órdenes altos)	Compromiso entre suavizado y selectividad
Paso-alto	Realce de bordes	Amplificación de ruido	Detección de contornos
Notch	Eliminación selectiva de frecuencias	Posibles distorsiones locales	Eliminación de ruido periódico

El diseño de filtros en el dominio de la frecuencia consiste en la definición de máscaras espectrales. Sin embargo, efectos en el dominio espacial, como *ringing* y desenfoque, emergen directamente de esas elecciones en el espectro.

5.6 *Wavelets* y Multirresolución

La Transformada de Fourier descompone la señal en frecuencias **globales**: cada coeficiente $F(u, v)$ recibe contribuciones de toda la imagen, sin información explícita sobre la ubicación espacial de esas frecuencias. Así, las estructuras localizadas, como los bordes, se representan de forma distribuida en el espectro.

Las ***wavelets* (ondículas)** superan esta limitación al utilizar funciones base **localizadas en el espacio**, que pueden ser desplazadas y escaladas. Estas funciones poseen **soporte compacto**, es decir, son diferentes de cero solo en una región finita del dominio, permitiendo una representación simultánea en términos de **frecuencia y ubicación espacial**.

5.6.1 El Límite de la Transformada de Fourier: localización espacial

La Transformada de Fourier describe con precisión **qué frecuencias están presentes** en una señal, pero no representa explícitamente **dónde ocurren esas frecuencias en el espacio**.

En el experimento presentado en la Figura 5.16, dos imágenes con estructuras localizadas en posiciones diferentes producen espectros de magnitud prácticamente idénticos. Esto ocurre porque la representación de Fourier es global: cada coeficiente recibe contribución de toda la imagen.

Como consecuencia, el espectro de magnitud no representa explícitamente la localización de bordes u otras estructuras, solo la distribución de las frecuencias presentes. Esta limitación motivó el desarrollo de representaciones multirresolución, como la Transformada *Wavelet* Discreta (DWT), capaces de describir simultáneamente la frecuencia y la localización espacial de las estructuras de la imagen.

```
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=["Señal A", "Espectro A", "Señal B (Desplazada)", "Espectro B"],
        cols=4, figsize=(14, 4)
        )
```

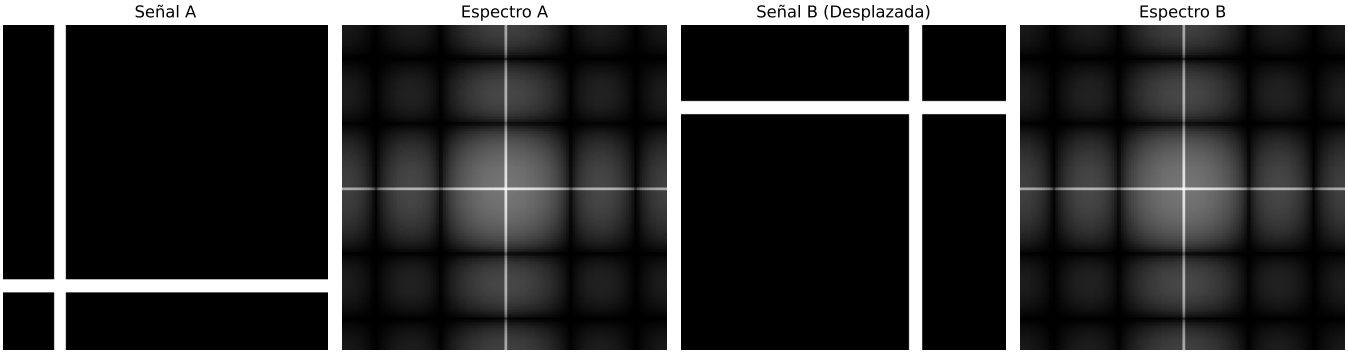


Figura 5.16: La transformada de Fourier global es ciega a la posición. Los espectros no dicen dónde están los bordes.

5.6.2 Transformada Wavelet Discreta 2D

La Transformada Wavelet Discreta (DWT) aplica, de manera separada en las direcciones horizontal y vertical, dos filtros complementarios: un **pasa-bajos** h (aproximación) y un **pasa-altos** g (detalles), seguidos de un submuestreo por un factor de 2 en cada dimensión. Este proceso produce cuatro subbandas, cuyos nombres indican la combinación de los filtros aplicados en cada dirección (L = *Low-pass*, pasa-bajos; H = *High-pass*, pasa-altos). Las características de cada subbanda se resumen en la Tabla 5.3.

$$\mathrm{DWT}(f) = \{ \underbrace{\mathrm{LL}}_{\text{aprox.}} , \underbrace{\mathrm{LH}}_{\text{detalles horizontales}} , \underbrace{\mathrm{HL}}_{\text{detalles verticales}} , \underbrace{\mathrm{HH}}_{\text{detalles diagonales}} \}.$$

Tabla 5.3: Subbandas producidas por la Transformada Wavelet Discreta 2D (DWT), indicando los filtros aplicados en cada dirección y el contenido predominante de cada componente.

Subbanda	Filtros aplicados	Contenido visual
LL	baja × baja	Aproximación de la imagen (versión suavizada y reducida)
LH	baja × alta	Bordes horizontales y variaciones verticales
HL	alta × baja	Bordes verticales y variaciones horizontales
HH	alta × alta	Detalles diagonales y texturas

La descomposición puede aplicarse recursivamente sobre la subbanda LL, generando una representación multirresolución. Tras J niveles, se obtiene una estructura con $3J + 1$ subbandas, donde cada nuevo nivel reduce la resolución de la componente de aproximación.

> **Conexión con CNNs**
>
> La descomposición multirresolución de las *wavelets* posee una relación conceptual con las representaciones jerárquicas utilizadas en redes neuronales convolucionales (CNNs). En ambos casos, sucesivas etapas de filtrado y reducción de resolución producen descripciones cada vez más abstractas de la imagen. Sin embargo, las ***wavelets* utilizan filtros matemáticamente definidos y reconstruibles**, mientras que las **CNNs aprenden sus filtros durante el entrenamiento**.

5.6.3 Familias de *Wavelets*

Diferentes familias de *wavelets* presentan compromisos distintos entre **soporte espacial**, suavidad y capacidad de compresión. El **soporte** corresponde a la extensión de la función *wavelet* en el dominio espacial: cuanto

menor es el soporte, más localizada está la función; cuanto mayor es, más suave tiende a ser su representación, aunque con un mayor costo computacional. La Tabla 5.4 compara algunas de las familias más utilizadas.

Tabla 5.4: Comparación entre familias de *wavelets*, destacando la longitud del soporte, el número de momentos nulos, la simetría y las aplicaciones típicas.

Wavelet	Longitud del soporte	Momentos nulos	Simetría	Uso típico
Haar	2	1	Asimétrica	Introducción y análisis básico
Daubechies db4	8	4	Asimétrica	Compresión y análisis general
Symlet sym4	8	4	Casi simétrica	Reconstrucción de señales
Biortogonal 5/3	5/3	2/2	Simétrica	JPEG 2000 sin pérdida
Biortogonal 9/7	9/7	4/4	Simétrica	JPEG 2000 con pérdida

Los **momentos nulos** miden la capacidad de la *wavelet* para representar regiones suaves de la imagen con pocos coeficientes distintos de cero. Una *wavelet* con p momentos nulos anula exactamente polinomios de grado hasta $p-1$. En consecuencia, cuanto mayor es el número de momentos nulos, mayor tiende a ser la eficiencia de compresión en regiones homogéneas, aunque esto generalmente implica funciones con un soporte más largo.

La Figura 5.17 presenta las funciones de base (*wavelets*) $\psi(t)$ en el dominio espacial. Estas funciones poseen **soporte compacto**, es decir, son distintas de cero solo en una región finita del dominio, a diferencia de las sinusoides de la Transformada de Fourier, que se extienden por todo el dominio.

```
#   Importar 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()
```

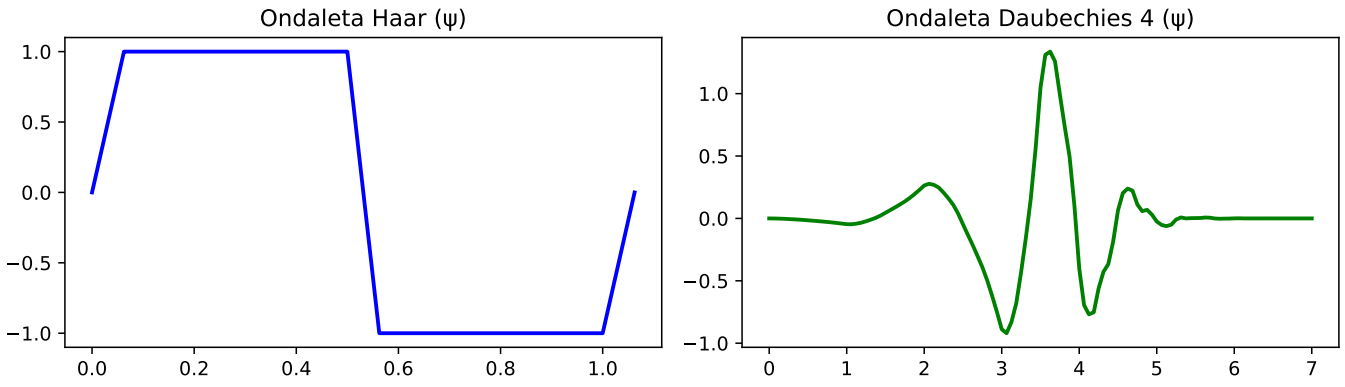


Figura 5.17: Funciones de la *Wavelet* ( ). Obsérvese cómo decaen rápidamente a cero (soporte compacto), a diferencia de las sinusoides infinitas de Fourier.

El diagrama de la Figura 5.18 ilustra el análisis multirresolución realizado por la DWT, en el cual la subbanda de aproximación (LL) se descompone sucesivamente, formando una representación jerárquica con dos niveles.

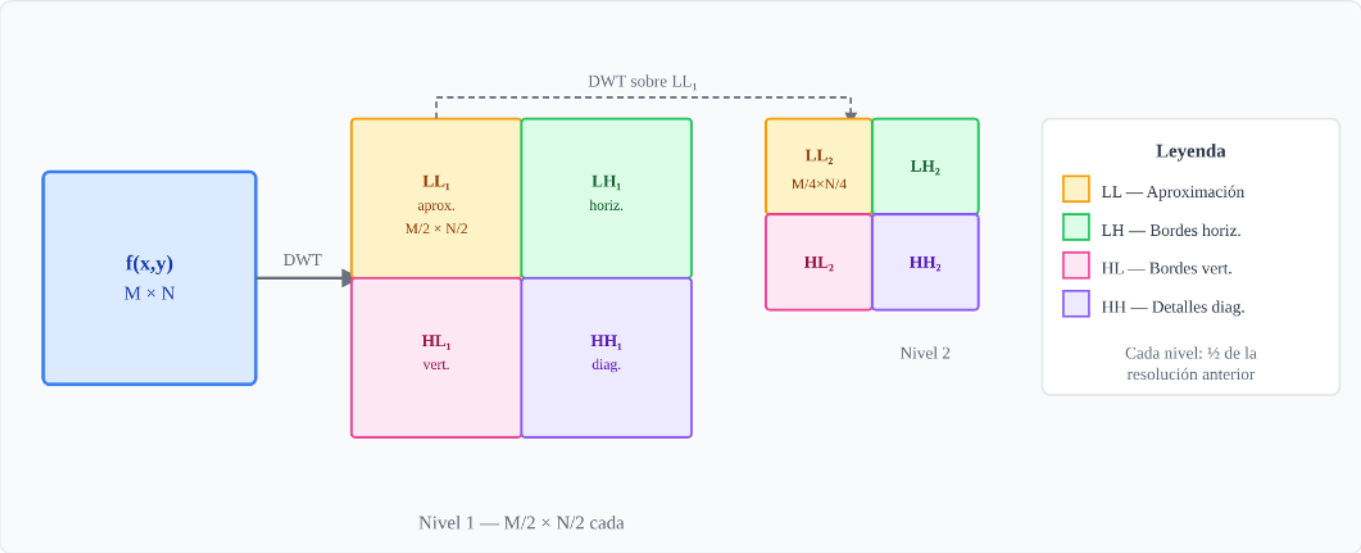


Figura 5.18: Diagrama de la descomposición *wavelet* 2D en dos niveles.

El simulador de la Figura 5.19 permite explorar interactivamente la Transformada *Wavelet* Discreta 2D (DWT) utilizando la *wavelet* de Haar. La descomposición en subbandas evidencia la separación entre la componente de aproximación y las componentes de detalle de la imagen.

Los diferentes patrones de entrada permiten observar el comportamiento direccional de los filtros. En imágenes con bordes horizontales y verticales, las subbandas LH y HL destacan, respectivamente, las variaciones verticales y horizontales de la intensidad. En regiones de variación suave, la mayor parte de la energía se concentra en la subbanda de aproximación LL, mientras que las subbandas de detalle presentan coeficientes cercanos a cero.

El análisis multirresolución también se puede observar al aumentar el número de niveles de descomposición. En este caso, solo la subbanda LL_1 se descompone nuevamente, dando origen a las subbandas LL_2, LH_2, HL_2 y HH_2, que forman el segundo nivel de la representación jerárquica.

En patrones formados por regiones homogéneas de gran extensión, como un degradado suave o un tablero compuesto por bloques grandes, la energía permanece predominantemente concentrada en la subbanda LL. En el degradado, esto ocurre porque las diferencias entre píxeles vecinos son pequeñas. En el tablero, por su parte, los píxeles poseen prácticamente la misma intensidad en el interior de cada bloque, de modo que solo las fronteras entre bloques producen coeficientes no nulos en las subbandas de detalle. Como estas fronteras

ocupan solo una pequeña fracción de la imagen, su contribución a la energía total permanece reducida.

Para posibilitar el análisis visual de estas variaciones sutiles, el simulador incorpora un control de ganancia de contraste de los detalles (que varía de 1 a 8). Este parámetro funciona como un factor de amplificación lineal aplicado exclusivamente a los coeficientes de las subbandas de detalle (LH, HL y HH) antes de su renderización en pantalla. En escenarios de transición suave (como el gradiente) o de uniformidad local (como el interior de los bloques del tablero), las diferencias numéricas calculadas por el filtro paso-alto de Haar resultan en coeficientes muy cercanos a cero, lo que haría que los cuadrantes correspondientes fueran oscuros e imperceptibles a simple vista. Al multiplicar estos valores por la ganancia, el simulador recupera visualmente las estructuras de alta frecuencia ocultas y resalta la orientación de los bordes remanentes.

El gráfico de energía por subbanda cuantifica esta distribución entre la componente de aproximación y las componentes de detalle, demostrando que la ganancia visual no altera la métrica original de la energía. En imágenes naturales, la mayor parte de la energía se concentra en la subbanda LL, mientras que las subbandas LH, HL y HH representan principalmente bordes, texturas y otras variaciones locales de la intensidad.

5.6.4 Análisis Multirresolución con la DWT 2D

La Transformada Wavelet Discreta 2D (DWT) descompone una imagen en componentes de aproximación y detalle, organizadas de forma jerárquica en diferentes escalas y orientaciones. Como las subbandas de detalle en imágenes naturales frecuentemente presentan coeficientes de bajo contraste, los ejemplos prácticos a continuación utilizan un patrón geométrico sintético generado en Python. Este enfoque replica el comportamiento del simulador de la Figura 5.19, haciendo visualmente explícitos los efectos del filtrado espacial y de la descomposición multirresolución.

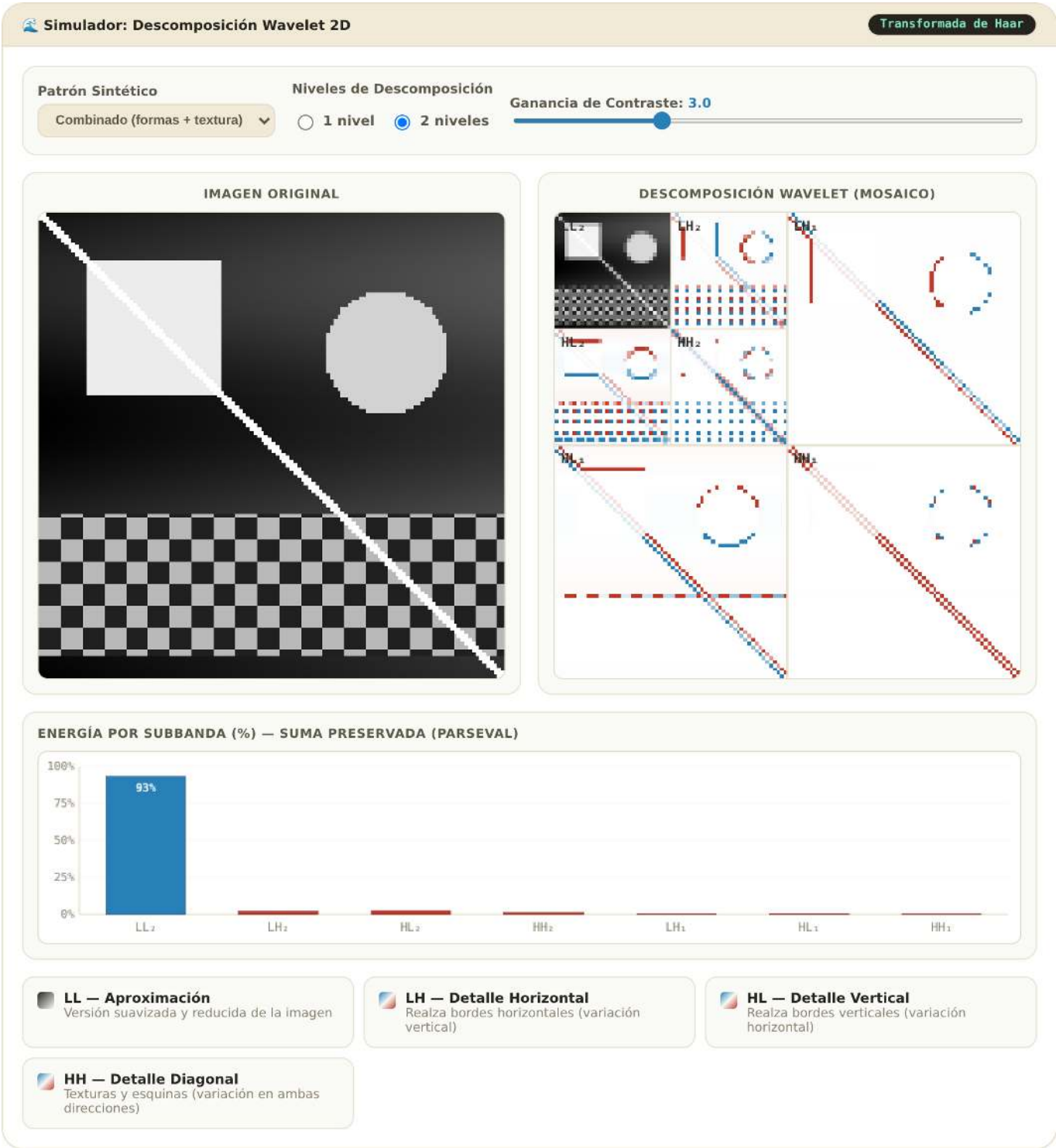
5.6.4.1 Descomposición en Mosaico de Múltiples Niveles

La Figura 5.20 ilustra la estructura jerárquica de la DWT en dos niveles utilizando la *wavelet* de Haar. El proceso se basa en la aplicación combinada de filtros pasa-baja y pasa-alta en las direcciones horizontal y vertical, seguidos por submuestreo por un factor de 2.

En el primer nivel, la imagen original origina la subbanda de aproximación (LL_1) y las componentes de detalle horizontal (LH_1), vertical (HL_1) y diagonal (HH_1). En el análisis multirresolución, la subbanda LL_1 se filtra y submuestrea nuevamente, generando el segundo nivel de descomposición (LL_2, LH_2, HL_2 y HH_2).

Para viabilizar la interpretación visual de las componentes de detalle, el código extrae el valor absoluto de sus coeficientes y aplica una normalización lineal (*min-max*) para ocupar toda la gama dinámica de tonos de gris [0, 255]. Esta operación transforma regiones homogéneas (coeficientes nulos) en negro y destaca en blanco los bordes y texturas extraídas en cada escala y orientación.

```
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

#   Generación de la Imagen Sintética (Mismo patrón 'combined' del simulador)
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)
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap05/sim-05-wavelet.html

Figura 5.19: Simulación de la descomposición *wavelet* 2D.

```
            # Cuadrado
            if 24 < x < 100 and 24 < y < 100:
                v = 225
            # Círculo
            cx, cy, r = 190, 76, 34
            if (x - cx)**2 + (y - cy)**2 < r**2:
                v = 205
            # Textura periódica (inferior)
            if y > 164 and y < 244:
                p = 12
                v = 185 if ((x // p + y // p) % 2 == 0) else 65
            # Línea diagonal
            if abs(x - y) < 4:
                v = 240
            img[y, x] = np.clip(v, 0, 255)
    return img.astype(np.uint8)

# Sustituye la imagen oscura de monedas por el patrón sintético claro
img_gray = gerar_imagem_sintetica(256)

#   Descomposición wavelet 2 niveles
wavelet   = "haar"
img_float = img_gray.astype(np.float64)

# Nivel 1
coefs1 = pywt.dwt2(img_float, wavelet)
LL1, (LH1, HL1, HH1) = coefs1

# Nivel 2 (aplicado sobre LL1)
coefs2 = pywt.dwt2(LL1, wavelet)
LL2, (LH2, HL2, HH2) = coefs2

def sb_vis(sb):
    """Normaliza subbanda para visualización [0,255]."""
    return cv2.normalize(np.abs(sb), None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

print(f"Forma original     : {img_gray.shape}")
print(f"LL1 (nivel 1)      : {LL1.shape}  |  LH1/HL1/HH1: {LH1.shape}")
print(f"LL2 (nivel 2)      : {LL2.shape}    |  LH2/HL2/HH2: {LH2.shape}")

imgs_dwt   = [img_gray, sb_vis(LL1), sb_vis(LH1), sb_vis(HL1), sb_vis(HH1),
              sb_vis(LL2), sb_vis(LH2), sb_vis(HL2), sb_vis(HH2)]
titles_dwt = ["Original",
              "LL  (aprox.)", "LH  (horiz.)", "HL  (vert.)", "HH  (diag.)",
              "LL  (aprox.)", "LH  (horiz.)", "HL  (vert.)", "HH  (diag.)"]

mm.show(imgs_dwt, titles=titles_dwt, cols=5, figsize=(16, 7))
```

```
Forma original     : (256, 256)
LL1 (nivel 1)      : (128, 128)  |  LH1/HL1/HH1: (128, 128)
LL2 (nivel 2)      : (64, 64)    |  LH2/HL2/HH2: (64, 64)
```

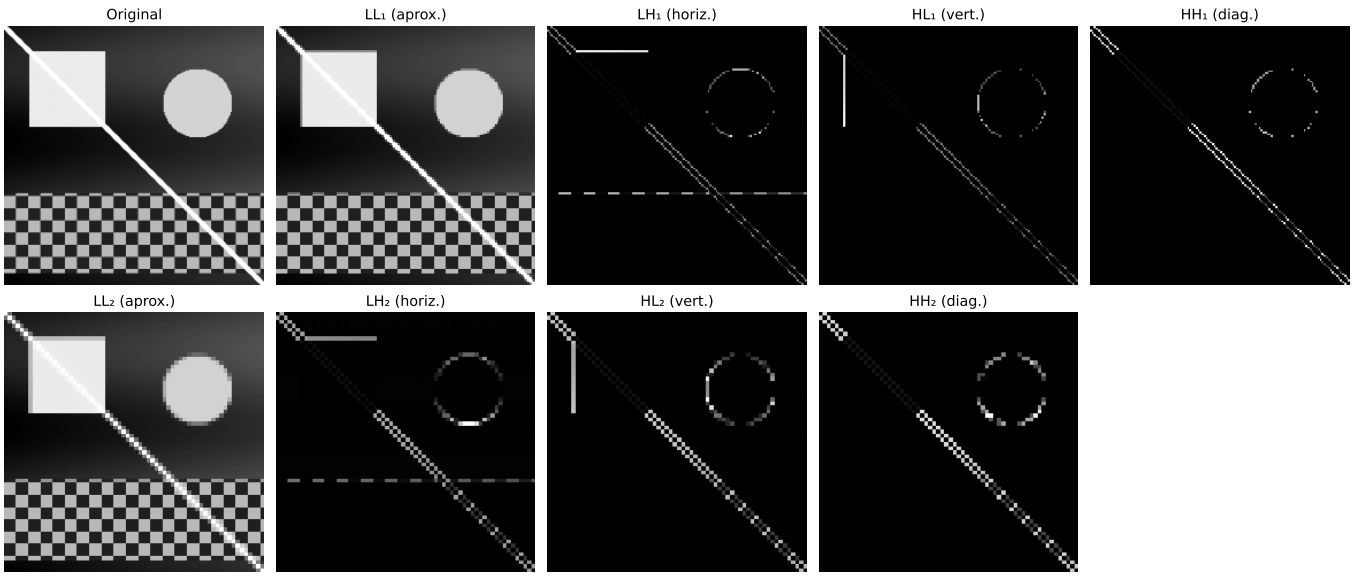


Figura 5.20: Descomposición *wavelet* 2D de 2 niveles con *wavelet* Haar: subbandas LL, LH, HL, HH en cada nivel. Las subbandas de detalle revelan estructuras orientadas en diferentes escalas utilizando un patrón sintético.

5.6.4.2 El Compromiso entre Localización y Suavidad

La elección de la función de base (*wavelet*) influye directamente en la forma en que las características de la imagen se distribuyen y codifican mediante los coeficientes de la DWT. La Figura 5.21 compara los resultados prácticos obtenidos al aplicar cuatro familias distintas sobre el patrón geométrico sintético: `haar`, `db4`, `sym4` y `bior2.2`.

Por poseer soporte corto y formato de función escalón, la *wavelet* de Haar produce coeficientes altamente localizados en las discontinuidades espaciales, generando bordes finos y nítidos en las subbandas de detalle. En contrapartida, familias como Daubechies (`db4`) y Symlets (`sym4`), que presentan mayor soporte (filtros más largos) y mayor número de momentos nulos, generan respuestas más suaves y distribuidas alrededor de las transiciones, lo que puede introducir ligeras oscilaciones o desenfoques en las fronteras abruptas.

Este comportamiento evidencia el clásico compromiso (*trade-off*) del análisis de multirresolución: soportes menores favorecen la localización espacial exacta de los bordes, mientras que soportes mayores y un mayor número de momentos nulos tienden a producir representaciones más dispersas y suaves. Esa suavidad y capacidad de atenuación de altas frecuencias garantizan una mayor eficiencia en la compactación de la energía, características fundamentales para aplicaciones de compresión de datos y eliminación de ruido (*denoising*).

```
import numpy as np
import cv2
import pywt

# Garantiza que img_gray e img_float utilicen el mismo patrón sintético claro
if 'gerar_imagem_sintetica' in globals():
    img_gray = gerar_imagem_sintetica(256)
else:
    # Fallback en caso de que el bloque anterior no se haya ejecutado en la misma sesión
    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))
```

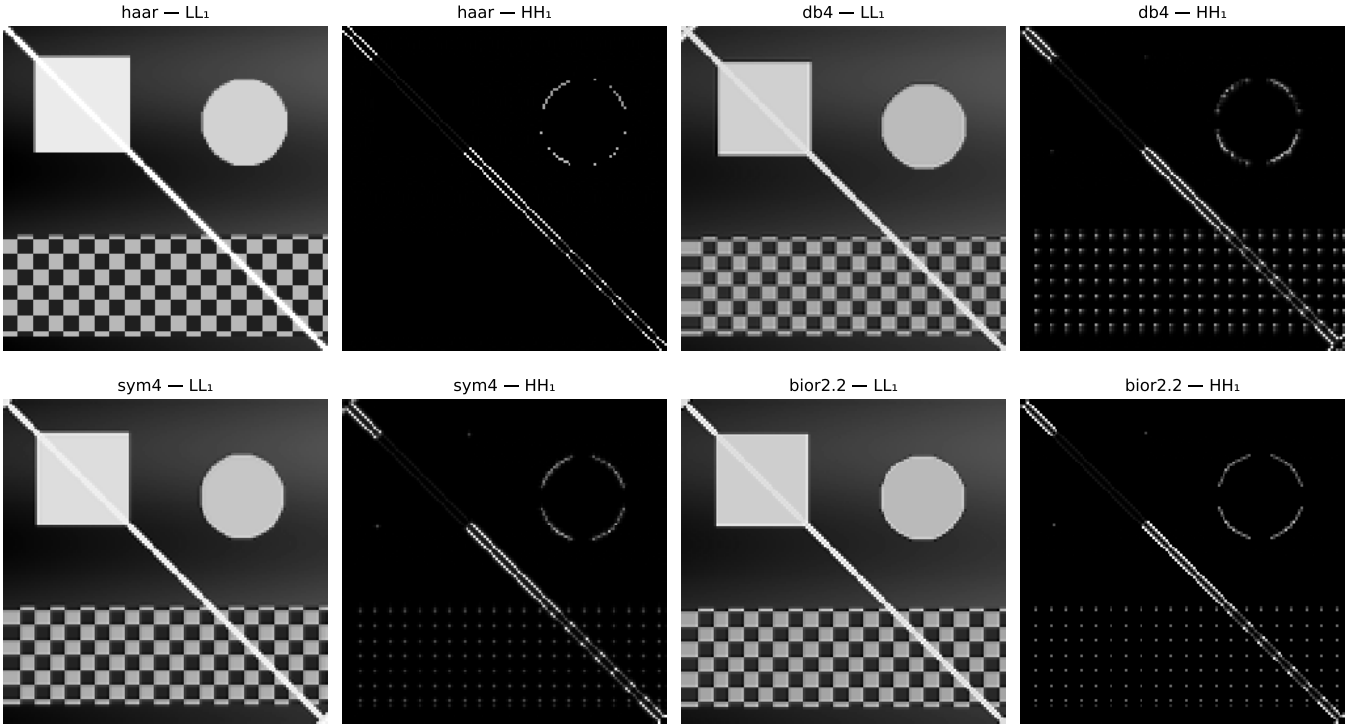


Figura 5.21: Comparación entre familias de *wavelets*: Haar, db4, sym4 y bior2.2. Subbanda LL (aproximación) y HH (diagonal) para cada elección, ilustrando el compromiso entre compactación y suavidad con base en el patrón sintético.

5.6.4.3 Limiarización de Coeficientes y Compresión

Una de las principales aplicaciones de la Transformada *Wavelet* Discreta (DWT) es la compresión de datos, impulsada por la capacidad de representación **dispersa** de los coeficientes. La Figura 5.22 ilustra el efecto de la limiarización abrupta (*hard thresholding*), técnica en la cual los coeficientes de detalle con magnitud inferior a un umbral T se anulan por completo antes del proceso de síntesis realizado por la Transformada *Wavelet* Discreta Inversa (IDWT).

A medida que se eleva el umbral T, un volumen creciente de coeficientes de alta frecuencia se pone a cero. Al concentrar menor energía, la eliminación de estos componentes reduce considerablemente la cantidad

de información necesaria para representar la imagen, manteniendo intacta la componente de aproximación global (la subbanda LL más profunda) para preservar la estructura macro. Visualmente, este descarte de coeficientes se manifiesta a través de la desaparición progresiva de texturas finas y el suavizado de transiciones abruptas de intensidad.

La fidelidad de la imagen reconstruida frente a la original se cuantifica mediante la métrica de **Pico de Relación Señal-Ruido (PSNR, *Peak Signal-to-Noise Ratio*)**, expresada en decibelios (dB). Valores más altos de PSNR indican menor distorsión y mayor proximidad matemática con la señal original. El experimento práctico evidencia la disminución gradual del PSNR conforme aumenta la agresividad de la limiarización, lo que permite evaluar numéricamente el umbral óptimo para el equilibrio entre compresión y degradación visual.

```
import numpy as np
import cv2
import pywt

# Garantiza que img_gray utilice el mismo patrón sintético claro
if 'gerar_imagem_sintetica' in globals():
    img_gray = gerar_imagem_sintetica(256)
else:
    # Respaldo en caso de que el bloque anterior no se haya ejecutado en la misma sesión
    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):
    """Descompone, aplica umbral y reconstruye vía IDWT."""
    coefs = pywt.wavedec2(img.astype(np.float64), wavelet, level=nivel)
    # Copia y aplica hard thresholding en todos los detalles
    coefs_t = [coefs[0]]   # LL final no se umbraliza
    for detalhe in coefs[1:]:
        coefs_t.append(tuple(pywt.threshold(sb, threshold, mode='hard') for sb in detalhe))
    rec = pywt.waverec2(coefs_t, wavelet)
    # Recorte a la dimensión original
    rec = rec[:img.shape[0], :img.shape[1]]
    return np.clip(rec, 0, 255).astype(np.uint8)

thresholds = [0, 10, 30, 60, 100]
imgs_thr   = [img_gray]
titles_thr = ["Original"]

for t in thresholds:
    rec  = dwt_threshold_reconstruct(img_gray, threshold=t)
    psnr = cv2.PSNR(img_gray, rec)
    imgs_thr.append(rec)
    titles_thr.append(f"T={t}  PSNR={psnr:.1f} dB")

mm.show(imgs_thr, titles=titles_thr, cols=3, figsize=(14, 10))
```

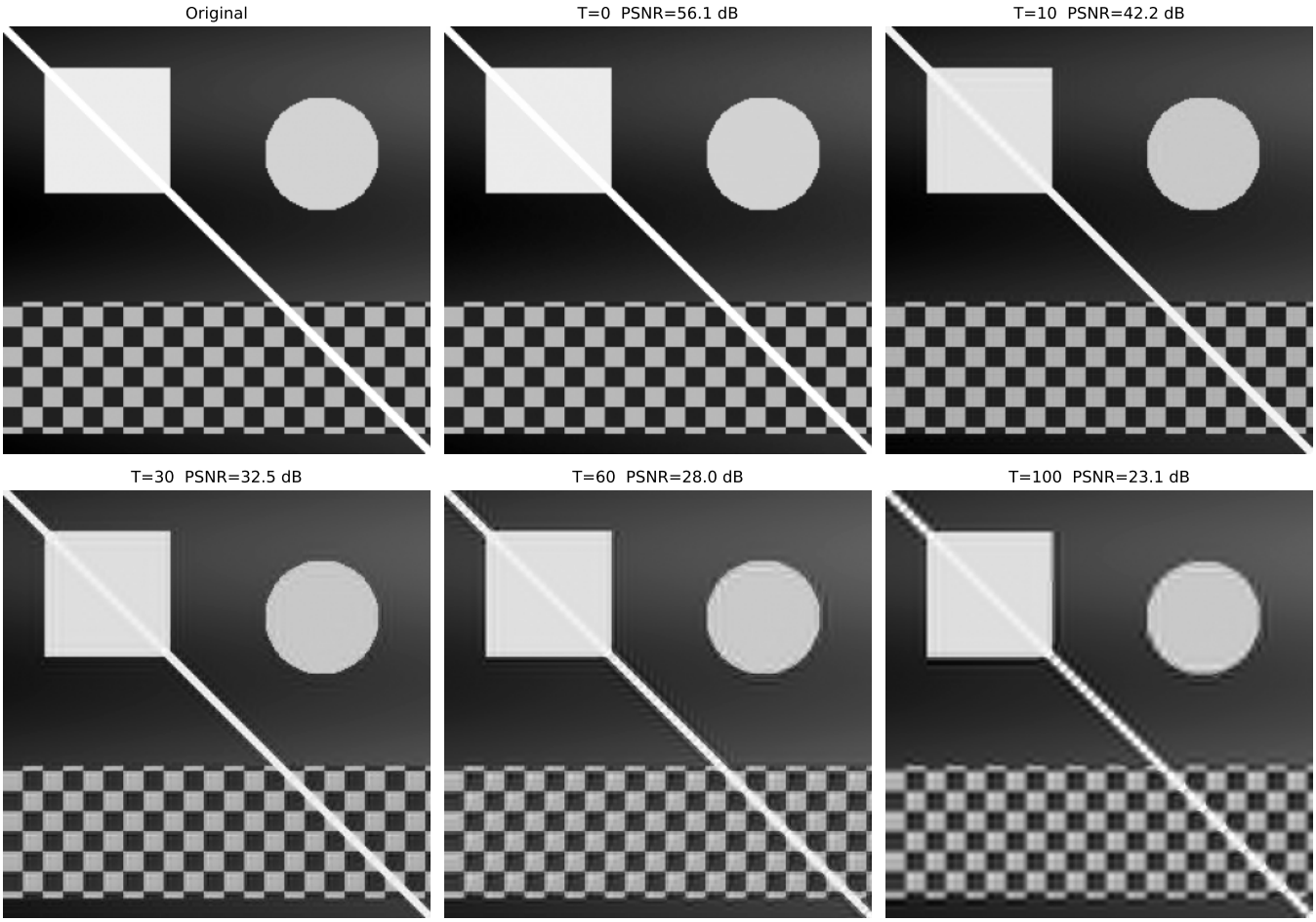


Figura 5.22: Reconstrucción *wavelet* con umbralización de coeficientes (*hard thresholding*): a medida que el umbral aumenta, más detalles se ponen a cero, produciendo imágenes progresivamente más suaves. La métrica PSNR cuantifica la pérdida de calidad sobre el patrón sintético.

Síntesis — Fourier vs. *Wavelets*: ¿cuándo utilizar cada enfoque?

La Tabla 5.5 sintetiza las principales diferencias estructurales y operativas entre la Transformada Discreta de Fourier (DFT) y la Transformada *Wavelet* Discreta (DWT).

Tabla 5.5: Comparación entre la Transformada Discreta de Fourier (DFT) y la Transformada *Wavelet* Discreta (DWT), destacando sus principales características y aplicaciones.

Criterio	Fourier (DFT)	*Wavelet* (DWT)
Funciones de base	Senoides de soporte infinito	Funciones de soporte compacto
Localización espacial	No explícita (global)	Explícita (local)
Filtrado espectral	Excelente para el control fino de frecuencias	Basado en subbandas (escalas)
Compresión de imágenes	Base de la DCT (JPEG tradicional)	Base de la DWT (JPEG 2000)
Análisis multiescala	No	Sí
Eliminación de ruido periódico	Altamente eficiente	Poco indicada
Señales no estacionarias	Limitada	Altamente eficiente

En términos prácticos, la DFT se consolida como la herramienta ideal para el análisis espectral puro, el diseño de filtros selectivos en el dominio de la frecuencia y la atenuación de ruidos periódicos y armónicos.

Por otro lado, la DWT sobresale en escenarios que exigen la preservación rigurosa de la localización espacial de las características asociada a su contenido frecuencial, destacándose en la compresión de datos, el análisis multirresolución y el procesamiento de transiciones abruptas. De este modo, ambas transformadas deben entenderse como técnicas perfectamente complementarias, que mapean caminos distintos y específicos para la resolución de problemas en PDI-VC.

Analogías con Audio: Limitaciones y Precauciones

Al establecer analogías entre el procesamiento de imágenes y el audio, es importante considerar las diferencias fundamentales:

- En los sistemas de audio estéreo/multicanal, la fase entre canales es crucial para la percepción de la localización espacial (diferencias interaurales de fase y tiempo).
- En los sistemas monoaurales, la fase tiene una influencia perceptual limitada: el oído humano es relativamente insensible a la fase absoluta de componentes sinusoidales aislados.
- En las imágenes, la fase de la DFT siempre es fundamental para la localización espacial de las estructuras, independientemente de que se trate de una imagen monocromática o en color.

La analogía entre la fase en audio y la fase en imágenes debe emplearse con cautela, destacando que, aunque ambas transportan información sobre la organización espacial/temporal de la señal, los mecanismos perceptuales son fundamentalmente diferentes.

5.7 Compresión de Imágenes

Mientras que las *wavelets* establecen la base teórica del estándar JPEG 2000, el estándar JPEG tradicional se basa en la **Transformada Discreta de Cosenos (DCT, *Discrete Cosine Transform*)**. A pesar de las diferencias estructurales, ambos enfoques comparten el mismo principio fundamental: compactar la energía de la imagen en un número reducido de coeficientes y descartar los componentes de menor relevancia con un impacto visual mínimo.

El objetivo central de la compresión es reducir el volumen de datos necesario para el almacenamiento o transmisión de una imagen. Este proceso es posible gracias a la identificación y eliminación de **redundancias** estructurales y perceptuales.

5.7.1 Taxonomía de las Redundancias

El desarrollo de algoritmos de compresión se fundamenta en la identificación y eliminación de tres categorías principales de redundancia, sintetizadas en la Tabla 5.6.

Tabla 5.6: Categorías de redundancia en imágenes digitales y sus respectivos mecanismos de explotación.

Tipo	Definición	Enfoque de Explotación
Espacial (interpíxel)	Alta correlación y dependencia estadística entre píxeles vecinos.	DCT, DWT y codificación predictiva.
Espectral (intercanal)	Correlación estadística entre los canales de color de una misma imagen.	Transformaciones de espacio de color (ej: RGB a YC_bC_r).
Psicovisual	Insensibilidad del sistema visual humano (SVH) a variaciones de alta frecuencia y bajo contraste.	Procesos de cuantización selectiva de coeficientes.

Dependiendo de la preservación de la información original tras el proceso de decodificación, los métodos de compresión se dividen en dos clases fundamentales:

- **Sin pérdida (*lossless*):** Garantiza una reconstrucción bit a bit idéntica a la imagen original. Se emplea en escenarios donde la integridad de los datos es estrictamente crítica, como en imágenes médicas, diagnósticos por imagen y almacenamiento de documentos textuales.
- **Con pérdida (*lossy*):** Admite la introducción de una distorsión controlada en la señal a cambio de tasas de compresión sustancialmente más elevadas. Es el enfoque estándar para fotografías de consumo y *streaming* de vídeo, ecosistemas en los cuales el SVH tolera pequeñas atenuaciones de alta frecuencia sin percepción de degradación de la calidad visual.

5.7.2 Transformada de Cosenos Discreta (DCT-II 2D)

La **Transformada de Cosenos Discreta** (DCT) constituye la operación central del estándar JPEG. A diferencia de la DFT, que utiliza una base compleja, la DCT se basa en funciones trigonométricas puramente reales. Para un bloque de imagen $f(x, y)$ de dimensiones $N \times N$, la **DCT-II 2D** mapea la señal espacial al dominio de las frecuencias espaciales, generando la matriz de coeficientes $C(u, v)$ mediante:

$$C(u, v) = \alpha(u)\,\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y)\, \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right] \tag{5.9}$$

donde los factores de normalización ortogonal están dados por $\alpha(0) = \sqrt{1/N}$ y $\alpha(k) = \sqrt{2/N}$ para $k > 0$.

Cada coeficiente $C(u, v)$ cuantifica la contribución —o "peso"— de una frecuencia espacial específica dentro de ese bloque. El término $C(0, 0)$ se denomina **componente DC** y representa la intensidad media del bloque (frecuencia nula). Los demás coeficientes, llamados **componentes AC** (*Alternating Current*), corresponden a las frecuencias espaciales progresivamente mayores.

5.7.3 Las Funciones de Base de la DCT

Desde una perspectiva geométrica, la Ecuación 5.9 realiza la proyección del bloque de píxeles sobre un conjunto de funciones ortogonales. Para el caso estándar de JPEG ($N = 8$), el bloque espacial se descompone en una combinación lineal de **64 funciones de base** bidimensionales, denotadas por $B_{u,v}(x, y)$ y generadas por el producto de funciones cosenoidales:

$$B_{u,v}(x, y) = \cos\left[\frac{\pi(2x+1)u}{16}\right] \cos\left[\frac{\pi(2y+1)v}{16}\right]$$

De esta manera, la operación inversa puede interpretarse como la reconstrucción exacta del bloque original mediante la suma ponderada de estas 64 matrices de base, donde cada coeficiente $C(u, v)$ actúa como el peso analítico de su respectiva componente armónica.

La **frecuencia espacial** indicada por los índices (u, v) determina el número de ciclos de oscilación a lo largo de las dimensiones horizontales y verticales del bloque. Como se ilustra en la Figura 5.23 —cuyo código aísla cada base aplicando la transformación inversa sobre impulsos unitarios—, estas 64 funciones se organizan en una matriz 8×8. La esquina superior izquierda ($u = 0, v = 0$) muestra el patrón uniforme de frecuencia nula (DC), mientras que el avance hacia la derecha (eje u) o hacia abajo (eje v) mapea variaciones armónicas progresivamente mayores, representando transiciones rápidas, bordes y texturas en las orientaciones horizontales, verticales y diagonales.

DCT vs DFT: Ventaja de la Compactación de Energía

Tanto la DCT como la DFT mapean un bloque espacial $N \times N$ en una matriz de coeficientes de la misma dimensión. Sin embargo, para imágenes naturales, la DCT presenta una mayor eficiencia en la **compactación de energía** en las bajas frecuencias. Esto ocurre porque la DCT asume implícitamente una simetría par de la señal en las fronteras del bloque, lo que equivale a una extensión periódica continua, minimizando el efecto de dispersión espectral (*ringing*). Como consecuencia, la mayoría de los

coeficientes AC decae rápidamente hacia valores cercanos a cero, optimizando el *pipeline* de compresión sin introducir degradación visual perceptible.

```
from scipy.fft import dct, idct  # línea añadida

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()
```

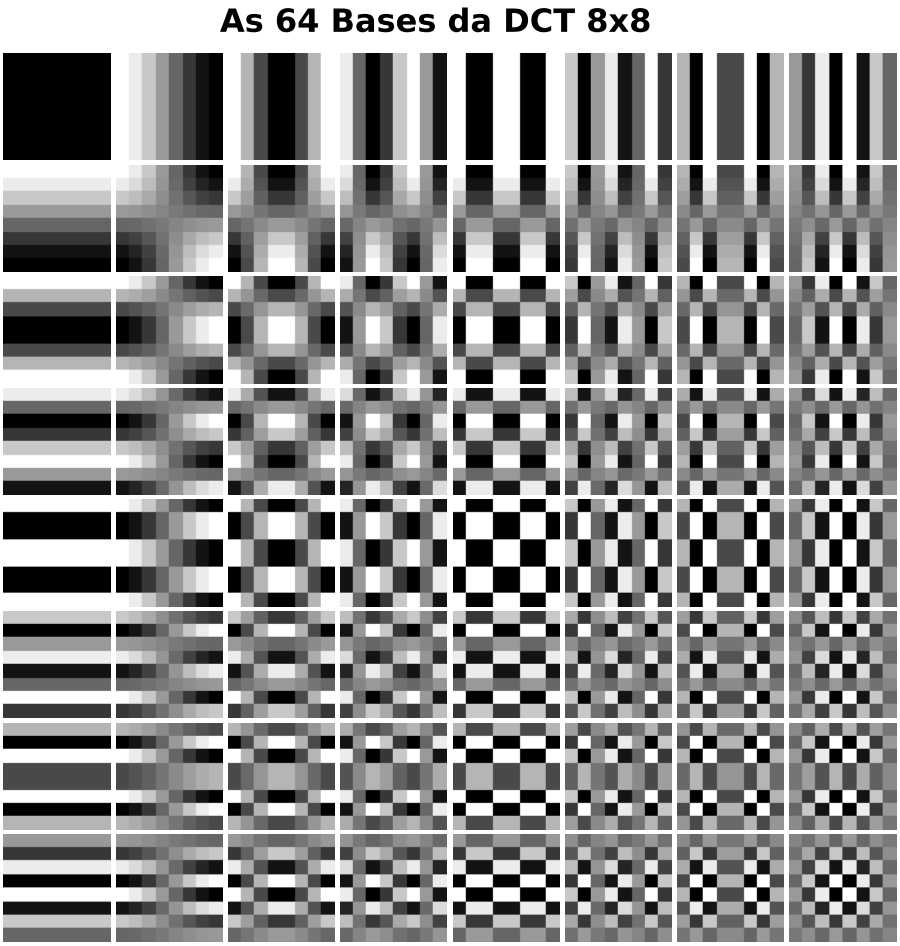


Figura 5.23: El Alfabeto Visual del JPEG: Las 64 funciones de base de la DCT-II. El coeficiente DC está en la parte superior izquierda (suave). Al descender y avanzar hacia la derecha, la oscilación espacial aumenta drásticamente.

5.7.4 Concentración de Energía y Reconstrucción Progresiva

Antes de la aplicación de la DCT, los píxeles del bloque de intensidad se trasladan rutinariamente (restando 128 para imágenes de 8 bits) con el fin de centrar la señal en torno a cero, eliminando componentes continuas innecesarias. Al calcular la DCT sobre el bloque resultante, la propiedad de **compactación de energía** se hace evidente: casi la totalidad de la varianza y de la información de la imagen original se concentra en el

coeficiente DC ($C(0,0)$) y en los primeros armónicos AC de baja frecuencia.

La Figura 5.24 demuestra este fenómeno mediante una reconstrucción progresiva por truncamiento abrupto. En lugar de utilizar los 64 coeficientes, el algoritmo preserva únicamente los k primeros componentes —seleccionados con base en un barrido que prioriza las bajas frecuencias espaciales— y anula los restantes.

La síntesis inversa (**IDCT**) realizada con solo una fracción de los coeficientes (como 15% o 30%) ya es capaz de recuperar las estructuras y la iluminación macro del bloque original de píxeles. A medida que los armónicos de frecuencias más altas se reincorporan progresivamente, los detalles finos y las transiciones rápidas se restauran. Este comportamiento valida el principio de la compresión perceptual: las altas frecuencias descartadas poseen poca energía y su ausencia, en condiciones normales, genera un impacto visual secundario en la percepción del observador.

```python
from scipy.fft import dct, idct

def dct2(bloco):
    """DCT-II 2D ortogonal (separable)."""
    return dct(dct(bloco.T, norm='ortho').T, norm='ortho')

def idct2(coefs):
    """IDCT-II 2D ortogonal."""
    return idct(idct(coefs.T, norm='ortho').T, norm='ortho')

#   Bloque 8×8 centralizado de la imagen
cy, cx = img_gray.shape[0]//2, img_gray.shape[1]//2
bloco  = img_gray[cy:cy+8, cx:cx+8].astype(np.float64) - 128.0

C = dct2(bloco)

print("Coeficientes DCT del bloque 8×8:")
print(np.round(C).astype(int))
print(f"\nEnergía DC     : {C[0,0]**2:.1f}")
print(f"Energía total  : {(C**2).sum():.1f}")
print(f"Fracción en DC : {C[0,0]**2 / (C**2).sum():.1%} ← concentración de energía")

#   Reconstrucción progresiva
imgs_rec  = [cv2.normalize((bloco+128).astype(np.uint8), None, 0, 255, cv2.NORM_MINMAX)]
titles_rec = ["Bloco original\n(8×8 pixels)"]

for keep in [1, 4, 10, 20, 40, 64]:
    C_trunc = np.zeros_like(C)
    indices = sorted([(u,v) for u in range(8) for v in range(8)], key=lambda p: p[0]+p[1])
    for u, v in indices[:keep]:
        C_trunc[u, v] = C[u, v]
    rec = np.clip(idct2(C_trunc) + 128, 0, 255).astype(np.uint8)
    imgs_rec.append(rec)
    titles_rec.append(f"{keep} coef.\n({keep/64:.0%} do total)")

mm.show(imgs_rec, titles=titles_rec, cols=4, figsize=(12, 7))
```

```
Coeficientes DCT del bloque 8×8:
[[ 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]]
```

```
Energía DC     : 202725.1
Energía total  : 639868.0
Fracción en DC : 31.7% ← concentración de energía
```

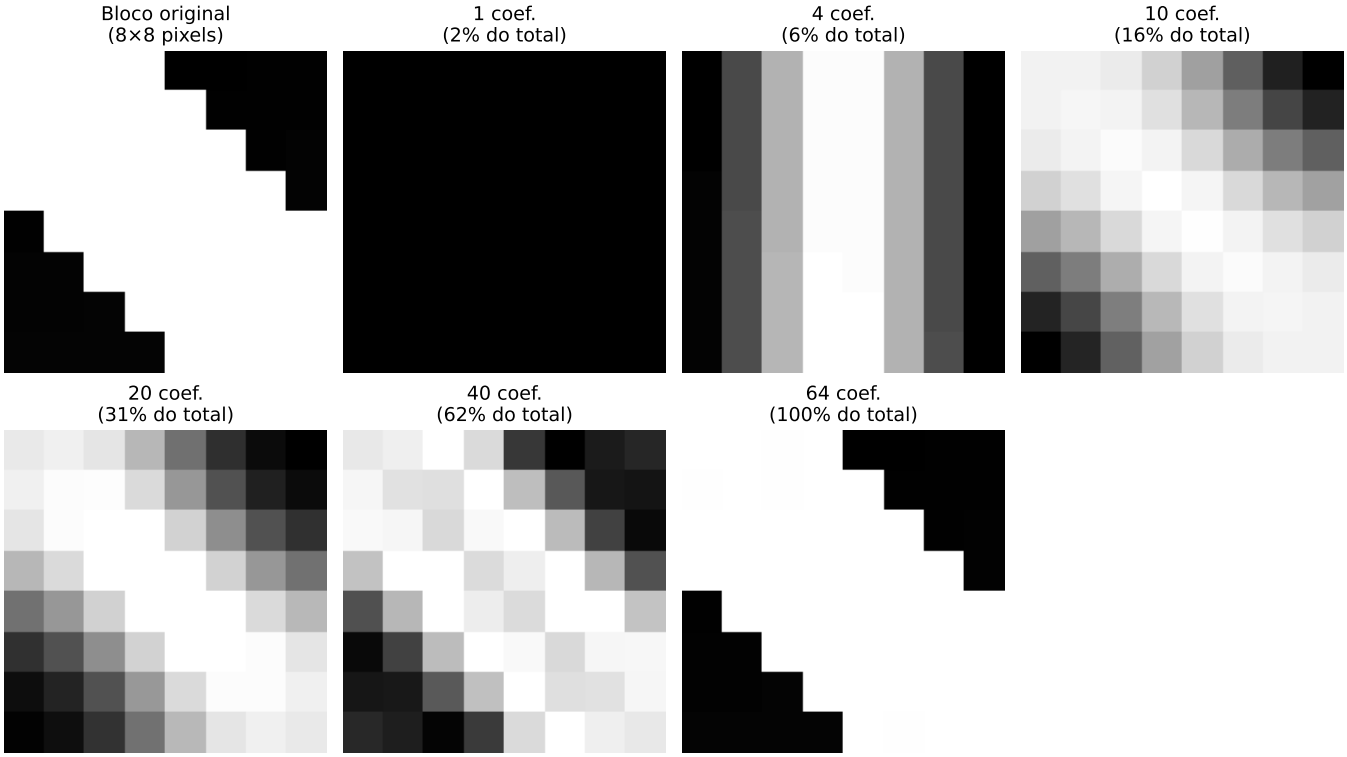


Figura 5.24: DCT 2D en bloque 8×8: coeficientes y reconstrucción progresiva.

5.7.5 El *Pipeline* de Compresión JPEG

El estándar JPEG opera dividiendo la imagen en bloques disjuntos de 8×8 píxeles, procesados mediante una secuencia de transformaciones espaciales, perceptuales y estadísticas. El *pipeline* completo de codificación se estructura en seis etapas principales:

$$\text{RGB} \xrightarrow{(1)\ YC_bC_r} \xrightarrow{(2)\ \text{Submuestreo}} \xrightarrow{(3)\ \text{Bloques } 8\times8} \xrightarrow{(4)\ \text{DCT}} \xrightarrow{(5)\ \text{Cuantificación}} \xrightarrow{(6)\ \text{Codificación Entrópica}}$$

La Tabla 5.7 detalla la función analítica y el fundamento perceptual que justifica cada una de estas etapas.

Tabla 5.7: Etapas del *pipeline* de compresión JPEG y sus respectivos fundamentos de diseño.

Etapa	Operación	Fundamento Perceptual y Estadístico
1	Conversión $RGB \to YC_bC_r$	Separa la luminancia (Y) de la crominancia (C_b, C_r). El sistema visual humano (SVH) presenta mayor sensibilidad a variaciones de brillo que de color.
2	Submuestreo de crominancia (ej: 4:2:0)	Reduce la resolución espacial de los canales de color a la mitad, descartando datos redundantes con impacto visual despreciable.

Etapa	Operación	Fundamento Perceptual y Estadístico
3–4	Centralización y aplicación de la DCT 8×8	Traslada los píxeles al intervalo $[-128, 127]$ y compacta la energía espectral del bloque en los coeficientes de baja frecuencia.
5	Cuantificación lineal selectiva	Divide cada coeficiente $C(u, v)$ por el elemento correspondiente de la matriz $Q(u, v)$, aplicando redondeo entero. Constituye la principal fuente de compresión con pérdida.
6	Barrido en zigzag y codificación	Ordena los coeficientes cuantificados para maximizar secuencias nulas consecutivas, optimizando la codificación por longitud de corrida (RLE) y la codificación de Huffman.

La **matriz de cuantificación** $Q(u, v)$ es el mecanismo central de control del compromiso entre tasa de compresión y calidad visual. En el algoritmo práctico de la Figura 5.25, el factor de calidad estipulado por el usuario (escala de 1 a 100) se convierte en un escalar que parametriza la severidad de la matriz Q. Valores reducidos de calidad expanden los divisores de $Q(u, v)$, forzando el truncamiento masivo de los coeficientes AC a cero. Cuando esta eliminación es excesiva, la discontinuidad en las fronteras de los bloques adyacentes no se atenúa en la reconstrucción, generando los denominados **artefactos de bloque** (*blocking artifacts*).

La Lógica del Barrido en Zigzag

La eficiencia del codificador entrópico posterior a la cuantificación depende directamente de la ordenación de los datos. Como la DCT concentra la energía vital en el vértice superior izquierdo de la matriz (bajas frecuencias) y empuja los coeficientes nulos hacia las extremidades opuestas, la lectura lineal por filas o columnas fragmentaría las secuencias de ceros.

La ordenación en zigzag soluciona esta limitación al recorrer la matriz diagonalmente en orden creciente de frecuencia espacial. Este mapeo agrupa los coeficientes significativos al inicio del vector y concentra los coeficientes nulos en una única secuencia continua al final del arreglo, permitiendo que el algoritmo RLE codifique grandes bloques de datos de forma compacta y eficiente.

i ¿Qué es RLE?

RLE (*Run-Length Encoding*) es una técnica de compresión sin pérdidas que codifica secuencias consecutivas de valores idénticos — especialmente **ceros** — como un par (recuento, valor). En JPEG, después del barrido en zigzag, los coeficientes cuantificados se organizan de modo que los ceros se concentren al final del vector. El RLE entonces comprime esa larga corrida de ceros con extrema eficiencia, optimizando el almacenamiento y la transmisión de la imagen comprimida.

```
import numpy as np
import cv2
from scipy.fft import dct, idct

#   Carga Segura de la Imagen de la Cámara (skimage)
try:
    from skimage import data
```

```
    img_gray = data.camera()
except ImportError:
    import subprocess
    subprocess.run(["pip", "install", "scikit-image", "-q"])
    from skimage import data
    img_gray = data.camera()

# Redimensiona ligeramente a 256x256 para mantener el patrón y velocidad de las pruebas
anteriores
img_gray = cv2.resize(img_gray, (256, 256))

#   Tabla de cuantización de luminancia (estándar 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):
    """DCT-II 2D ortogonal (separable)."""
    return dct(dct(bloco.T, norm='ortho').T, norm='ortho')

def idct2(coefs):
    """IDCT-II 2D ortogonal."""
    return idct(idct(coefs.T, norm='ortho').T, norm='ortho')

def jpeg_compress_block(bloco, Q_table):
    """DCT → cuantización → de-cuantización → IDCT en bloque 8×8."""
    C  = dct2(bloco.astype(np.float64) - 128)
    Cq = np.round(C / Q_table) * Q_table    # cuantiza y de-cuantiza
    return np.clip(idct2(Cq) + 128, 0, 255)

def jpeg_quality_compress(img, qualidade=50):
    """JPEG simplificado: comprime imagen completa por bloques 8×8."""
    if qualidade < 50:
        escala = 5000 / qualidade
    else:
        escala = 200 - 2 * qualidade
    # Corregido de 'scala' a '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)

#   Comparación de factores de calidad
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)
```

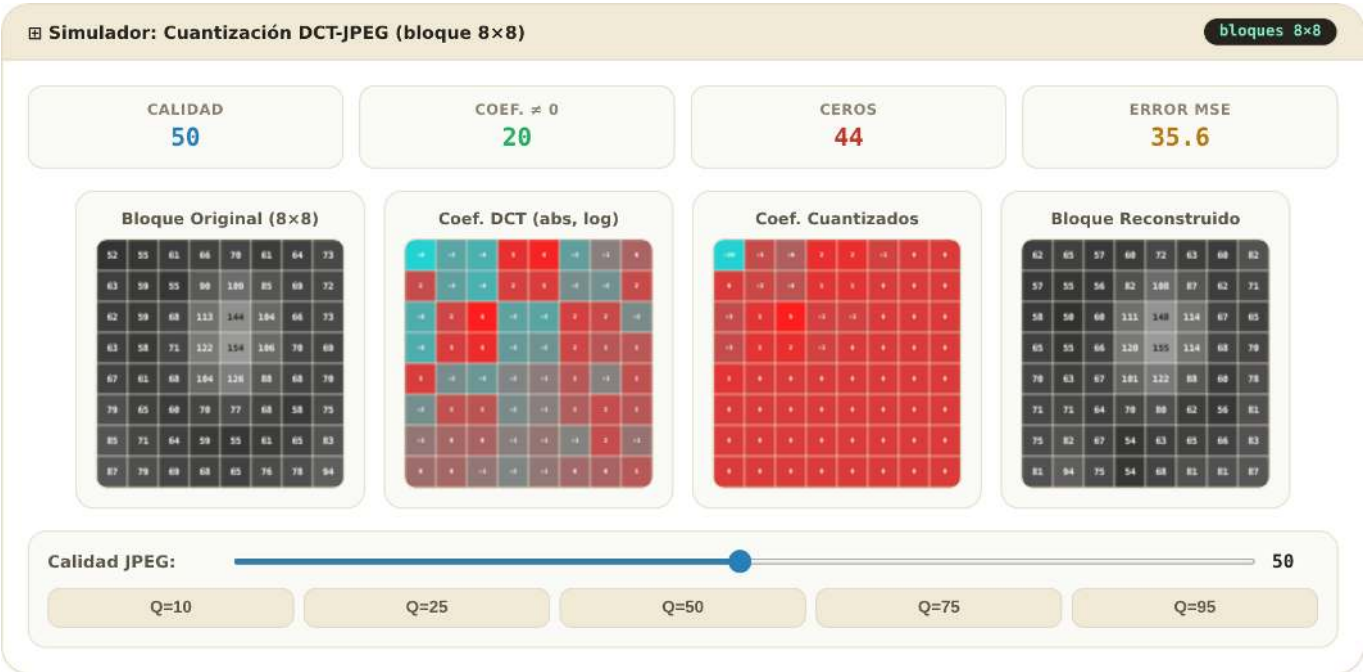
```
    imgs_jpeg.append(rec)
    titles_jpeg.append(f"Q={q}\nPSNR={psnr:.1f}dB")

mm.show(imgs_jpeg, titles=titles_jpeg, cols=3, figsize=(14, 10))
```

Figura 5.25: *Pipeline* JPEG simplificado aplicado a la imagen clásica del *Cameraman*: DCT en bloques 8×8, cuantización con diferentes factores de calidad y reconstrucción vía IDCT. Los artefactos de bloque (*blocking artifacts*) se vuelven visualmente evidentes en factores de calidad reducidos ($Q = 10$ y $Q = 25$).

5.7.6 Simulador Interactivo: Cuantización DCT

El simulador de la Figura 5.26 permite explorar el impacto del proceso de cuantización sobre un bloque 8×8 extraído de una imagen real, sintetizando en tiempo real los siguientes componentes:

- **Bloque original y reconstruido:** Representación directa de los píxeles en el dominio espacial en escala de grises [0, 255].
- **Coeficientes DCT:** Distribución de la energía mapeada de forma logarítmica en un gradiente cromático, evidenciando la concentración de intensidad en el vértice superior izquierdo (bajas frecuencias).
- **Coeficientes cuantizados:** Exhibición de los valores enteros resultantes de la división por la matriz $Q(u, v)$, haciendo visualmente explícita la aparición masiva de coeficientes nulos (en tonos oscuros) conforme el factor de calidad se reduce.
- **Métricas de compresión:** Panel de monitoreo que cuantifica el Error Cuadrático Medio (MSE), el número de coeficientes preservados y el volumen de ceros generados para la codificación entrópica.



Versión interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap05/sim-05-dct.html

Figura 5.26: Simulador interactivo de compresión DCT-JPEG: ajuste el factor de calidad y visualice en tiempo real los coeficientes anulados, el bloque reconstruido y el error de cuantización.

5.8 Comparación de Formatos de Imagen

La elección de un formato de almacenamiento digital impacta directamente en el compromiso entre calidad visual, tamaño de archivo y costo computacional de decodificación. Los tres formatos de mayor relevancia para arquitecturas *web* y sistemas de computación visual son JPEG, PNG y WebP.

5.8.1 Características de los Formatos

La Tabla 5.8 sintetiza las propiedades estructurales de los principales formatos de imagen rasterizados.

Tabla 5.8: Comparación estructural entre los principales formatos de imagen rasterizados.

Característica	JPEG	PNG	WebP
Compresión	Con pérdida	Sin pérdida	Con y sin pérdida.
Transparencia (canal alfa)	No	Sí	Sí.
Soporte de animación	No	Limitado (APNG)	Sí.
Algoritmo base	DCT + Huffman	DEFLATE (LZ77 + Huffman)	VP8 / VP8L.
Mejor para	Fotografía	Gráficos, texto e iconos	Uso universal en entorno Web.
Peor para	Texto y bordes nítidos	Imágenes fotográficas complejas	Compatibilidad heredada.

5.8.2 Métricas de Evaluación de Calidad

Dos métricas objetivas son ampliamente adoptadas para cuantificar la distorsión introducida por procesos de compresión:

Pico de la Relación Señal-Ruido (PSNR, *Peak Signal-to-Noise Ratio*):

$$\text{PSNR} = 10\,\log_{10}\left(\frac{L^2}{\text{MSE}}\right) \quad [\text{dB}] \tag{5.10}$$

donde $L = 255$ para imágenes cuantizadas en 8 bits y MSE representa el **Error Cuadrático Medio** (*Mean Squared Error*). Valores de PSNR superiores a 40 dB indican excelente fidelidad; entre 30 dB y 40 dB representan buena calidad; y valores inferiores a 30 dB corresponden a degradaciones visuales fácilmente perceptibles.

Índice de Similitud Estructural (SSIM, *Structural Similarity Index*):

$$\text{SSIM}(f, g) = \frac{(2\mu_f\mu_g + c_1)(2\sigma_{fg} + c_2)}{(\mu_f^2 + \mu_g^2 + c_1)(\sigma_f^2 + \sigma_g^2 + c_2)} \tag{5.11}$$

El SSIM evalúa ventanas locales de la imagen basándose en tres componentes complementarios: **luminancia** (μ_f, μ_g), **contraste** (σ_f, σ_g) y **estructura** (σ_{fg}), ponderados por constantes de estabilidad c_1 y c_2. El índice varía en el intervalo $[-1, 1]$, donde la unidad representa la identidad perfecta. A diferencia del PSNR, el SSIM considera la organización espacial de los errores, alineándose con la percepción del sistema visual humano (SVH).

PSNR vs SSIM: Aplicación de Métricas Perceptuales

El PSNR posee una formulación matemática simple y un bajo costo computacional; sin embargo, tiende a sobreestimar la calidad en imágenes con distorsiones localizadas o subestimarla en variaciones globales de brillo toleradas por el observador. El SSIM modela con mayor fidelidad la percepción biológica, pero exige un mayor esfuerzo de procesamiento. Para análisis rigurosos de codificadores, se recomienda reportar ambas métricas estadísticas en carácter complementario.

5.8.3 Inspección Visual: Naturaleza de los Artefactos de Compresión

La naturaleza matemática del codificador determina el tipo de degradación introducida en tasas de bits reducidas. Como se ilustra en la Figura 5.27, la compresión agresiva mediante DCT en el estándar JPEG segmenta la imagen en mallas rígidas, generando los **artefactos de bloque** (*blocking artifacts*). En contrapartida, los algoritmos basados en codificación predictiva o representaciones sometidas a transformadas espaciales avanzadas (como WebP y JPEG 2000) eliminan las discontinuidades de bloque, pero introducen pérdida de textura fina y desenfoques característicos alrededor de bordes de alto contraste.

```
import os
import cv2

# Garantiza la existencia del directorio y guarda los archivos comprimidos
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])

# Extracción de región de interés para visualización de artefactos (Zoom de 4x)
zoom_original = cv2.resize(img_gray[120:200, 150:230], (320, 320),
                           interpolation=cv2.INTER_NEAREST)

rec_jpeg = cv2.imread("imagens/comp_test/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=["Zoom Original", "JPEG Q=10 (Artefacto de Bloque)", "WebP Q=10 (Suavizado)"],
        cols=3, figsize=(14, 5))
```

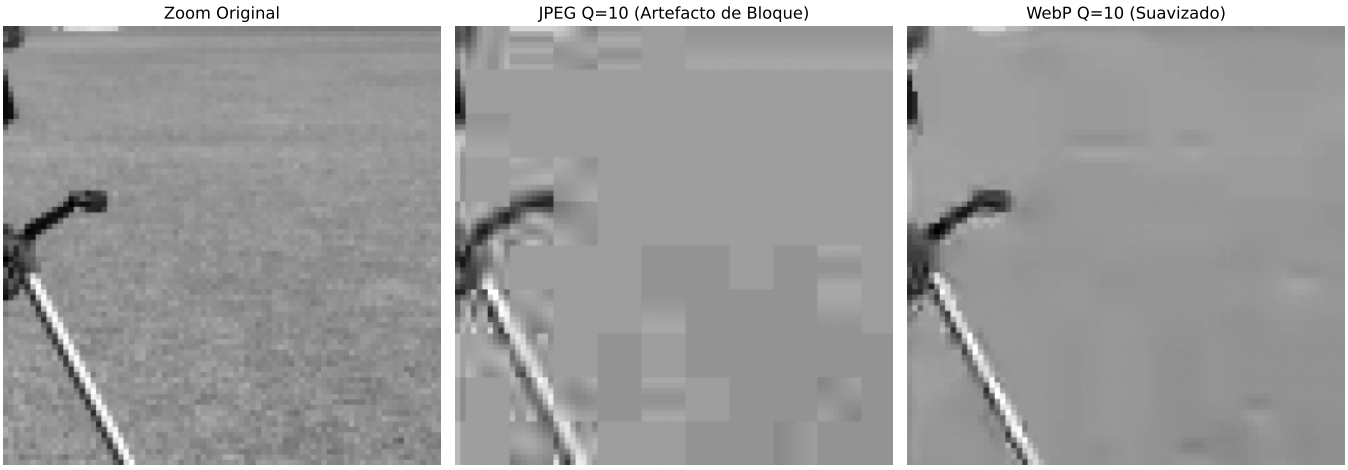


Figura 5.27: Análisis comparativo de artefactos de compresión bajo factor de calidad reducido ($Q = 10$). A la izquierda, se observa el artefacto de bloque característico de la discretización por DCT en JPEG. A la derecha, se evidencia el efecto de atenuación y suavizado de bordes intrínseco al estándar WebP.

5.8.4 Evaluación Cuantitativa y Espacial de la Compresión

La validación de los algoritmos de compresión con pérdida exige un análisis que correlacione el costo de almacenamiento con la fidelidad de la señal reconstruida. Esta evaluación se realiza de manera complementaria mediante curvas de rendimiento global y mediante el mapeo local de las distorsiones inducidas por los codificadores.

5.8.4.1 Curvas de Tasa-Distorsión

La Figura 5.28 presenta la evaluación empírica del *pipeline* JPEG y WebP mediante **curvas de tasa-distorsión**, que monitorean la ganancia de compresión (tamaño del archivo en KB) en función del PSNR. El formato PNG actúa como línea de base ideal (PSNR $= \infty$), ya que su naturaleza *lossless* impide cualquier degradación, aunque requiere un volumen de datos sustancialmente mayor.

El análisis de las curvas demuestra la superioridad y la eficiencia del estándar WebP sobre el JPEG tradicional: para alcanzar un mismo nivel de fidelidad matemática (como el rango de calidad excelente, donde PSNR > 40 dB), el codificador WebP genera archivos significativamente más pequeños. Este comportamiento refleja el impacto práctico de la evolución de los algoritmos en la optimización de los sistemas de transmisión y almacenamiento digital.

```
import os
import cv2
import matplotlib.pyplot as plt

# Garantiza la existencia del directorio de pruebas
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})

#   Generación de la Curva Tasa-Distorsión
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"\nTamaño bruto (sin compresión): {img_gray.nbytes/1024:.0f} KB")
print(f"\n{'Formato':>8} {'Qual.':>6} {'KB':>7} {'PSNR (dB)':>11}")
print("-"*38)
for r in resultados:
    psnr_s = f"{r['PSNR']:>11.2f}" if r['PSNR']!=float('inf') else f"{'∞ (lossless)':>11}"
    print(f"{r['formato']:>8} {str(r['qualidade']):>6} {r['KB']:>7.1f} {psnr_s}")
```

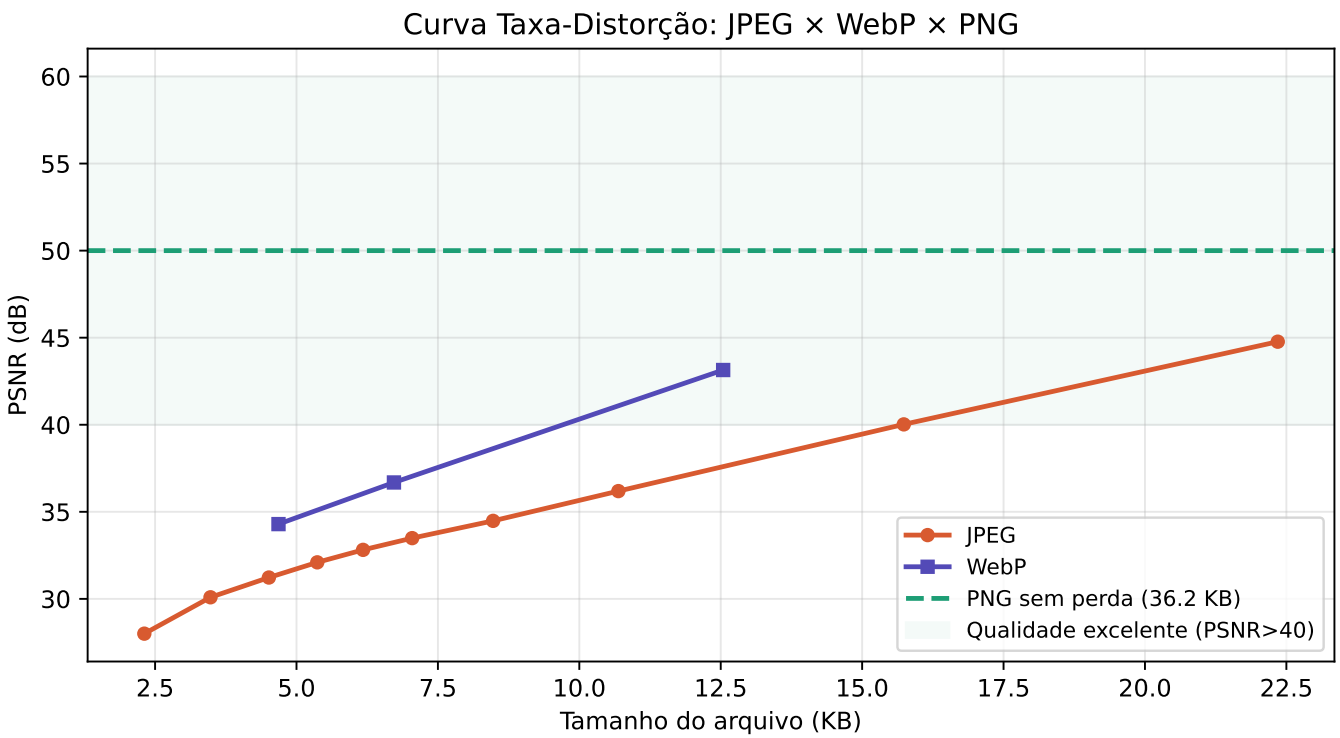


Figura 5.28: Curva tasa-distorsión: PSNR vs tamaño de archivo para JPEG, WebP y PNG aplicada a la imagen del *Cameraman*.

```
Tamaño bruto (sin compresión): 64 KB

 Formato  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
```

Tamaño original de la imagen

La imagen *Cameraman* (256 × 256 píxeles en escala de grises) ocupa **64 KB** en formato bruto (sin compresión). Como referencia, el PNG *lossless* comprime ese volumen a **36,2 KB** — evidenciando que la compresión sin pérdidas ya reduce significativamente el almacenamiento para imágenes con regiones homogéneas. En contrapartida, los formatos con pérdida (JPEG y WebP) alcanzan tamaños aún menores: el JPEG con calidad 95 ocupa 22,3 KB (PSNR 45 dB), mientras que el WebP con calidad 90 alcanza 12,5 KB con PSNR equivalente, demostrando su superioridad en eficiencia de compresión.

5.8.4.2 Mapeo Espacial de Errores y Correlación Perceptual

Aunque el PSNR ofrece un indicativo numérico rápido, las métricas globales no logran discriminar cómo se distribuye geométricamente la pérdida de información sobre la imagen. La Figura 5.29 soluciona esta limitación al asociar las reconstrucciones en diferentes calidades con sus respectivos mapas de error absoluto y con el SSIM.

Los mapas residuales — obtenidos mediante la diferencia absoluta normalizada entre la imagen original y la comprimida — revelan la firma espacial intrínseca de cada arquitectura de codificación:

- **En calidades altas ($Q = 95$ a $Q = 75$):** Las distorsiones se concentran predominantemente alrededor de transiciones abruptas de intensidad (bordes), como resultado del reflejo espectral derivado del descarte de altas frecuencias. El índice SSIM permanece cercano a la unidad, atestiguando la integridad de las estructuras originales.
- **En calidades agresivas ($Q = 50$ a $Q = 25$):** El error adopta una estructura de malla ortogonal regularizada. Este patrón geométrico evidencia la aparición de los **artefactos de bloque** (*blocking artifacts*), indicando que la cuantización severa ha corrompido la correlación espacial entre bloques adyacentes de 8×8 píxeles.

El SSIM captura esta degradación morfológica de manera mucho más sensible que el PSNR, penalizando la puntuación final a medida que la organización estructural y las texturas finas — a las cuales el sistema visual humano es altamente receptivo — son eliminadas por el codificador.

```
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

# Garantiza la existencia del directorio de pruebas
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"

    # GRABACIÓN FORZADA: Genera y graba el JPEG con la calidad actual en la ruta correcta
    img_compactada = jpeg_quality_compress(img_gray, qualidade=q)
    cv2.imwrite(path, img_compactada)

    # Lectura segura del archivo recién grabado
    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)
```

```
    # Diferencia absoluta normalizada para evidenciar la estructura espacial del 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))
```

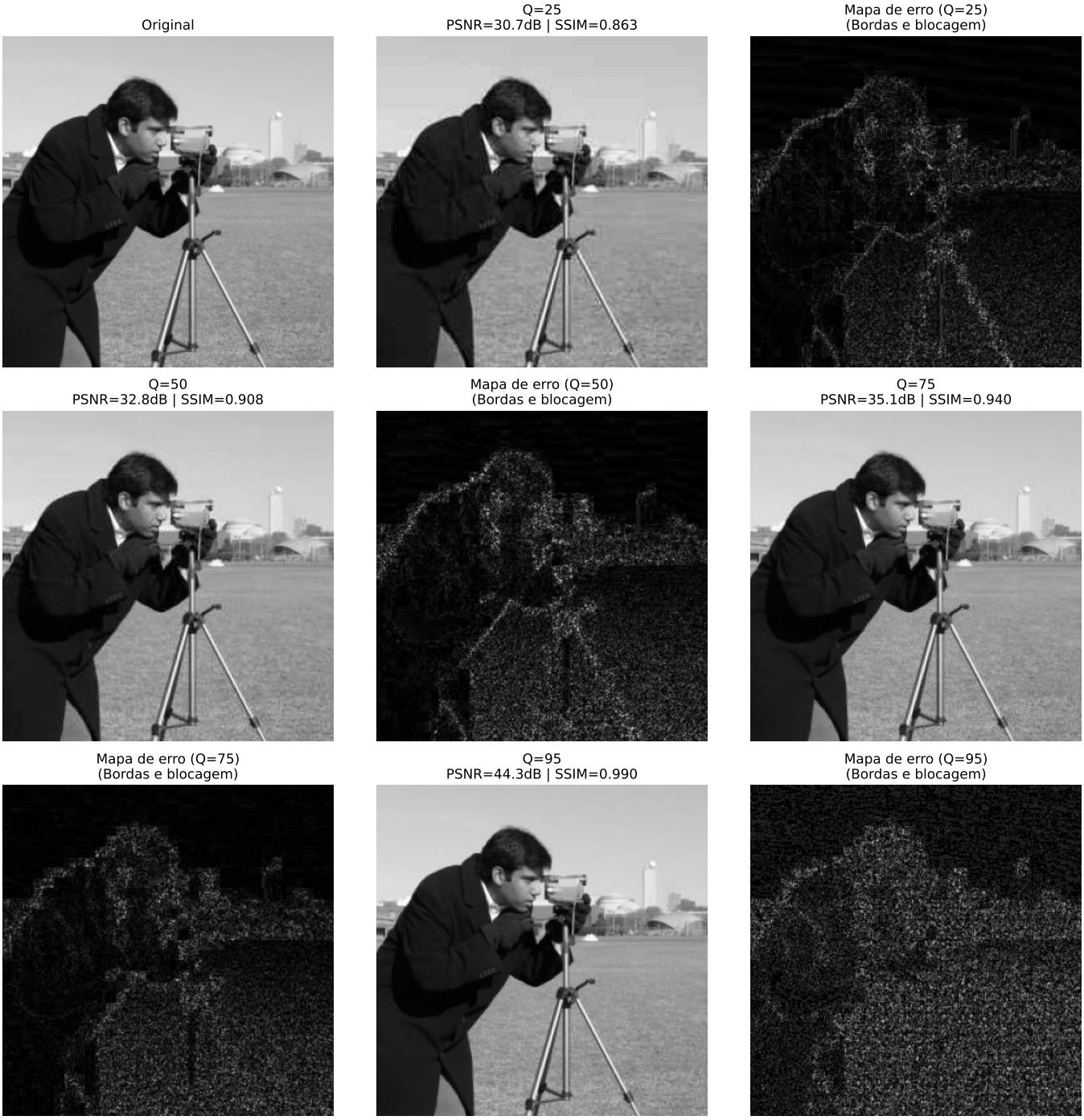


Figura 5.29: Análisis espacial de degradación: imágenes reconstruidas y sus respectivos mapas de error absoluto normalizados para diferentes factores de calidad JPEG.

Interpretando los mapas de error

Los mapas de error presentados fueron **normalizados individualmente** (`cv2.NORM_MINMAX`) para maximizar el contraste visual y revelar la estructura espacial de las distorsiones. Esto significa que:

- En **Q=95**, el error absoluto es del orden de **0.5–1.5 niveles de gris** (imperceptible visualmente), pero la normalización lo amplifica a blanco y negro para evidenciar su ubicación en bordes y transiciones.
- En **Q=25**, el error absoluto es **10–20 veces mayor** (5–15 niveles de gris), pero la normalización también lo lleva al mismo intervalo [0, 255].

Por lo tanto, **la intensidad del blanco en los mapas NO es comparable entre diferentes calidades** — los mapas sirven únicamente para revelar la **firma espacial** del error (bordes vs bloques), no su magnitud. La magnitud correcta está dada por los valores de PSNR y SSIM, que muestran claramente que Q=95 tiene un error mucho menor que Q=25.

Síntesis — Compresión JPEG

El proceso de compresión en el estándar JPEG se basa en la aplicación combinada de transformaciones espaciales, perceptuales y estadísticas para reducir las redundancias de una imagen. La Tabla 5.9 resume el papel de cada etapa en el *pipeline* y su respectivo impacto en la reducción de datos.

Tabla 5.9: Síntesis de las etapas del *pipeline* de compresión JPEG y sus respectivos impactos.

Etapa	Operación Analítica	Mecanismo de Ganancia / Compresión
Conversión YC_bC_r	Aislamiento de los canales de luminancia y crominancia.	Modela la percepción del SVH, permitiendo tratar el color y el brillo de forma independiente.
Submuestreo 4:2:0	Reducción de la resolución espacial de los canales de color (C_b y C_r).	Elimina aproximadamente el 50% de los datos brutos con un impacto visual mínimo.
DCT 8×8	Mapeo del dominio espacial al dominio de frecuencias espaciales.	Compactación de energía, concentrando la información vital en los primeros coeficientes.
Cuantización Lineal	División entera de los coeficientes por una matriz de ponderación $Q(u, v)$.	Principal fuente de compresión con pérdida; elimina altas frecuencias imperceptibles.
Codificación Entrópica	Aplicación de algoritmos RLE y codificación de Huffman.	Compresión estadística sin pérdida, optimizada por las largas series de coeficientes nulos.

Artefactos de Degradación Característicos

La aplicación de tasas de compresión excesivamente agresivas (factores de calidad reducidos) introduce distorsiones predecibles en la imagen reconstruida, derivadas de las limitaciones matemáticas del modelo:

- **Artefactos de bloqueo (*blocking artifacts*):** Discontinuidades geométricas visibles en los límites de los bloques de 8×8 píxeles, causadas por la pérdida de correlación espacial tras la cuantización severa de los componentes de CA.
- **Efecto de dispersión (*ringing*):** Oscilaciones fantasma o distorsiones de "humo" alrededor de bordes nítidos y de alto contraste, provocadas por la eliminación abrupta de armónicos de alta frecuencia necesarios para reconstruir funciones escalón.
- **Pérdida de textura fina:** Atenuación de detalles de alta frecuencia y bajo contraste (como céspedes, tejidos o porosidad), lo que hace que regiones originalmente texturizadas adopten un aspecto excesivamente liso u homogeneizado.

5.9 Aplicación Práctica: Eliminación de Ruido mediante Filtrado Híbrido

Reuniendo las técnicas consolidadas a lo largo de este capítulo, se presenta un *pipeline* completo de **restauración de imágenes** que combina el análisis espectral en el dominio de la frecuencia con el filtrado adaptativo en el dominio espacial. El objetivo es atenuar un ruido mixto (compuesto por degradación gaussiana e interferencia periódica) preservando al máximo los detalles estructurales de la imagen original.

$$\text{Imagen Ruidosa} \xrightarrow{\text{FFT2}} \xrightarrow{\text{Filtro Notch Gaussiano}} \xrightarrow{\text{IFFT2}} \xrightarrow{\text{Filtro Bilateral}} \text{Imagen Restaurada}$$

Evaluación Complementaria: PSNR vs. SSIM

El par de métricas estadísticas PSNR y SSIM proporciona una evaluación cualitativa y morfológica complementaria del proceso de restauración:

- **PSNR:** Penaliza uniformemente la desviación cuadrática media píxel a píxel.
- **SSIM:** Evalúa la preservación de estructuras locales perceptualmente relevantes (luminancia, contraste y contornos).

En la práctica, existe un compromiso analítico (*trade-off*) entre **reducción de ruido** y **preservación de detalles**: los filtros espaciales excesivamente agresivos atenúan bien el ruido de alta frecuencia, pero degradan texturas finas y suavizan bordes nítidos — lo que **reduce simultáneamente** tanto el PSNR como el SSIM en relación con la imagen original. El desafío del diseño de filtros es encontrar el punto de equilibrio que maximice ambas métricas, garantizando una restauración fiel y visualmente agradable.

5.9.1 Análisis de Rendimiento y Conclusión del Capítulo

Los resultados numéricos y visuales generados por la Figura 5.30 demuestran la relevancia práctica de asociar diferentes dominios de procesamiento. La inserción simultánea de ruido periódico y estocástico corrompe las propiedades morfológicas de la señal, reduciendo severamente los índices de similitud y la relación señal-ruido de la imagen de referencia.

El aislamiento y la supresión de los picos armónicos en el dominio de la frecuencia mediante la máscara *notch* eliminan las franjas de interferencia senoidales dispersas sobre el espacio bidimensional. Como se evidencia en los datos impresos de la Figura 5.30, este filtrado quirúrgico promueve un salto inmediato y sustancial en la métrica PSNR. No obstante, el ruido Gaussiano de alta frecuencia permanece activo de forma homogénea en el espectro, lo que exige un enfoque complementario.

La restauración final se consolida en el dominio espacial con la introducción del filtro bilateral. A diferencia de los operadores de paso bajo convencionales (como el Gaussiano o el de media), que suavizarían indiscriminadamente el ruido y los contornos estructurales, el filtrado bilateral calcula pesos ponderados por la proximidad geométrica y por la diferencia de intensidad radiométrica. Este comportamiento adaptativo atenúa las fluctuaciones estocásticas remanentes en las regiones de transición suave y preserva la nitidez de los bordes espaciales.

La convergencia de ambos enfoques resulta en una **mejora sustancial y simultánea** del PSNR y del SSIM en relación con la imagen ruidosa — aunque los valores finales permanecen inferiores a los de la imagen original (PSNR $= \infty$, SSIM $= 1{,}0$), debido a la pérdida inevitable de información espectral y de textura durante los procesos de filtrado. La atenuación suave (gaussiana) de los picos en el espectro evita los artefactos de *ringing*, mientras que el filtro bilateral elimina el ruido estocástico residual sin comprometer la nitidez de los bordes. Los resultados confirman la eficacia y la complementariedad práctica de las herramientas de análisis de frecuencia presentadas en este capítulo, demostrando que el filtrado híbrido (frecuencia + espacial) es superior a cualquier enfoque aislado para la restauración de imágenes degradadas por ruido mixto.

```
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. Construcción del ruido mixto
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. Espectro e identificación de los picos
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. Notch gaussiano en los picos periódicos
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. Filtro Bilateral: eliminación del ruido gaussiano residual
img_den = cv2.bilateralFilter(img_notch, d=7, sigmaColor=25, sigmaSpace=7)

#   Cálculo de las Métricas de Validación
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 visibles)",
        "Máscara notch\n(gaussiana suave)",
        f"Después de 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
```

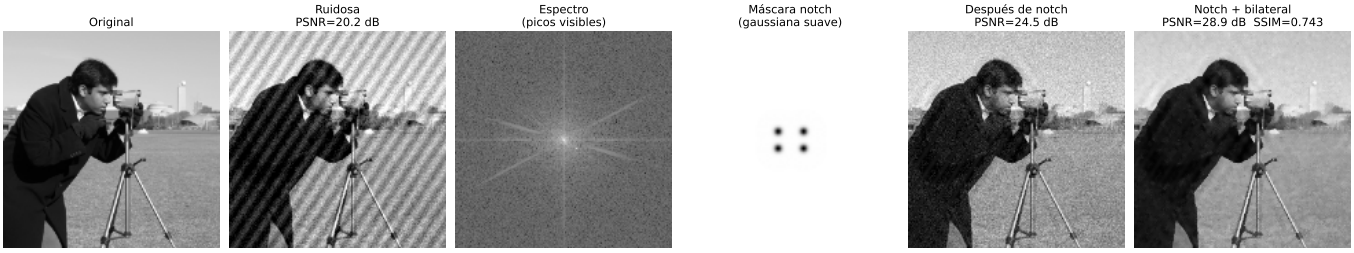


Figura 5.30: *Pipeline* completo de eliminación de ruido mixto: (1) adición de ruido gaussiano y periódico; (2) identificación de picos de interferencia en el espectro de frecuencias; (3) aplicación de máscara *notch* con atenuación gaussiana suave; (4) post-procesamiento mediante filtro bilateral para la eliminación del ruido estocástico residual.

5.10 Resumen del Capítulo

La transición del **dominio espacial** al **dominio de la frecuencia** revela la distribución espectral de energía de la imagen, estableciendo la base analítica para el filtrado avanzado, la restauración y la compresión de datos. La articulación estructural de estos conceptos se sintetiza en el mapa conceptual de la Figura 5.31.

Fundamentos Esenciales

- **DFT y Percepción Visual:** El espectro descompone la imagen en componentes armónicas. La **fase** retiene la inteligibilidad geométrica de la escena y la localización de contornos, mientras que la **magnitud** dicta la distribución de contraste y las amplitudes globales.
- **Eficiencia Algorítmica:** El Teorema de la Convolución hace viable el procesamiento de máscaras de gran escala en el dominio de la frecuencia mediante FFT, reduciendo la complejidad computacional asintótica de $O(N^2K^2)$ en el espacio a $O(N^2 \log N)$.
- **Fenómeno de *Ringing*:** Los cortes abruptos en el espectro (Filtros Ideales) generan oscilaciones espaciales no deseadas (fenómeno de Gibbs). La atenuación suave mediante filtros de **Butterworth** o **Gaussianos** elimina estas discontinuidades.
- **Análisis Multirresolución mediante *Wavelets*:** Superando el carácter puramente global de Fourier, la DWT captura la frecuencia y la localización espacial simultáneamente, fundamentando el estándar JPEG 2000 y subsidiando representaciones jerárquicas análogas a las extracciones de características en Redes Neuronales Convolucionales (CNNs).
- **Compresión Perceptual (DCT):** El *pipeline* JPEG explora las limitaciones de contraste del sistema visual humano en altas frecuencias espaciales. La DCT aísla la energía de bloques 8×8, permitiendo que la cuantización descarte coeficientes AC de detalles finos sin perjuicio perceptual severo.

Próximos Pasos: El **Capítulo 6** inaugura la Parte II de la obra, aplicando las herramientas de procesamiento de imágenes en la resolución de problemas reales de inspección industrial. Se explorarán técnicas de **segmentación y análisis de formas** para la detección automática de fallas en líneas de producción — desde la identificación de defectos superficiales en piezas hasta la lectura *QRCode* en pruebas, consolidando el puente entre la teoría presentada en la Parte I y las demandas prácticas de la visión computacional.

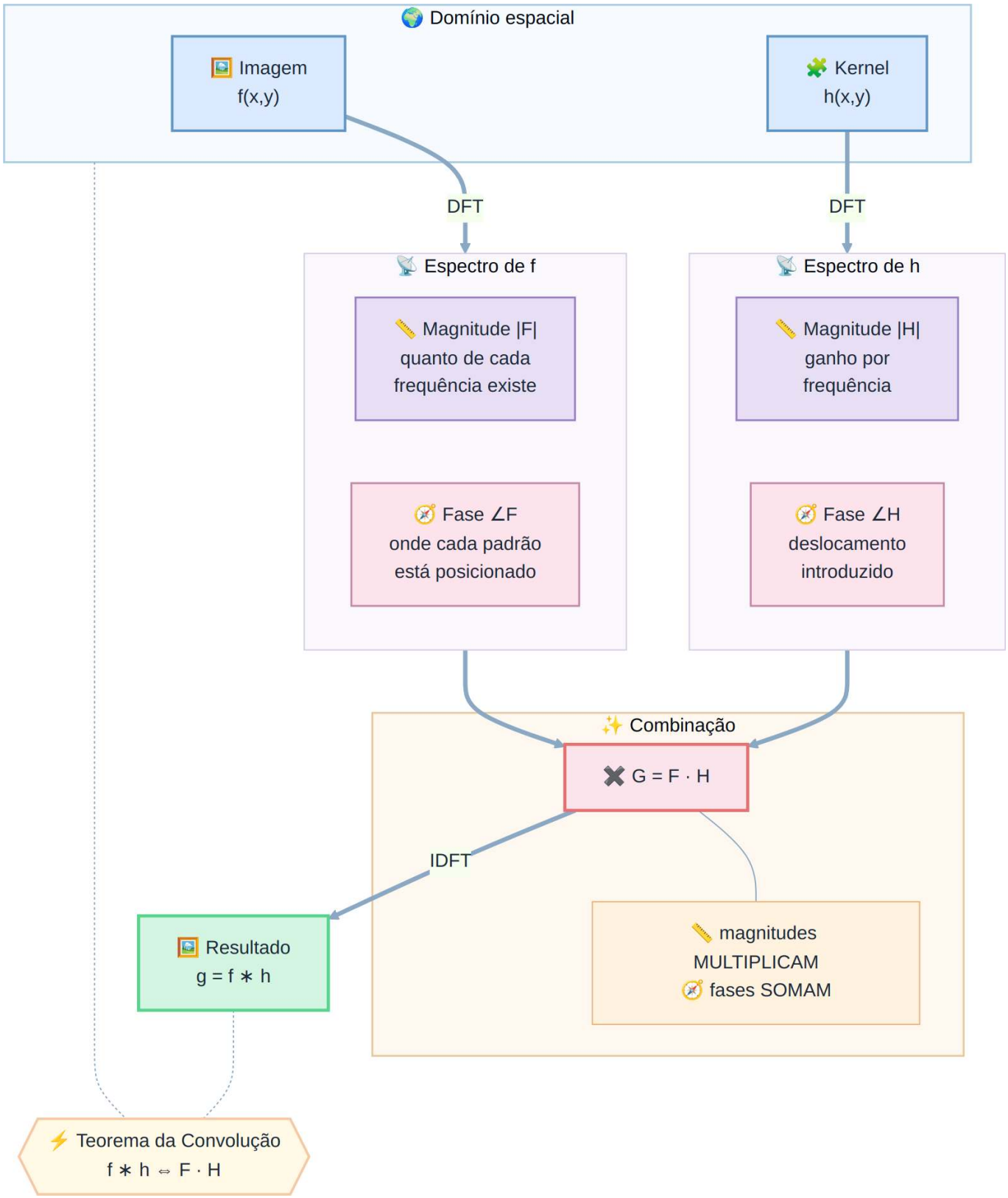


Figura 5.31: Mapa conceptual de las transformaciones y propiedades en el dominio de la frecuencia.

5.11 🤖 Uso de Gemini Notebook como Tutor Complementario

En esta edición, se incentiva el uso de la plataforma **Gemini Notebook** como herramienta complementaria de aprendizaje — **no como sustituta** de la lectura atenta, de la resolución de ejercicios o de la experimentación práctica. Basado en arquitecturas de inteligencia artificial, el sistema utiliza exclusivamente el material didáctico y los documentos proporcionados por el autor como base de conocimiento, asegurando que las respuestas generadas estén conceptualmente alineadas con el contenido programático y con el enfoque pedagógico adoptado a lo largo de esta obra.

Acceso al Tutor Inteligente

🚀 ACCEDER A Gemini Notebook: CAPÍTULO 05

🌐 **Idioma y Lenguaje de Programación**

El proyecto de este capítulo en Gemini Notebook fue construido únicamente con el texto en **portugués** y los ejemplos de código en **Python**. Si estás estudiando con la edición en inglés o francés, o siguiendo la ruta en C++, las respuestas del tutor pueden no corresponder exactamente con la versión que estás leyendo.

Directrices sobre el Contenido Generado por Inteligencia Artificial

Aunque las herramientas de inteligencia artificial constituyen aliados eficientes en el proceso de aprendizaje y revisión, el contenido generado está sujeto a inconsistencias o imprecisiones técnicas. Por ello, es indispensable la consulta sistemática de libros de texto, artículos científicos y fuentes académicas indexadas para la validación rigurosa de la información. Se recomienda encarecidamente la ejecución y la modificación de los ejemplos prácticos en Python proporcionados en este capítulo como método primario de verificación experimental de los resultados.

5.12 Lista de Ejercicios

1. **(10%) Implementación Directa de la DFT 2D:** Implemente analíticamente la Transformada Discreta de Fourier 2D (DFT) sin la ayuda de funciones nativas de bibliotecas (como `np.fft.fft2`), utilizando estrictamente la formulación matemática definida en la Ecuación 5.1 para una matriz de dimensiones 16×16. Realice la validación numérica comparando los coeficientes generados con los resultados de la función `np.fft.fft2`, asegurándose de que la desviación absoluta máxima sea inferior a 10^{-8}. Mida los tiempos de ejecución de ambos métodos y presente una justificación teórica para la disparidad observada en términos de complejidad asintótica.

2. **(15%) Supresión de Ruido Periódico:** Agregue interferencias sinusoidales con frecuencias espaciales $(u_0, v_0) \in \{(5, 10), (20, 5), (30, 30)\}$ a la imagen de prueba del *Cameraman*. Para cada escenario de degradación, diseñe una máscara de filtrado *notch* específica en el dominio de la frecuencia para aislar y atenuar los picos armónicos no deseados. Evalúe cuantitativamente la eficacia del proceso de restauración mediante el cálculo de las métricas de PSNR y SSIM. Discuta analíticamente el compromiso (*trade-off*) entre la atenuación del ruido sinusoidal y la indeseada atenuación de características estructurales legítimas de la imagen.

3. **(15%) Análisis Comparativo de Operadores Pasa-Bajas:** Realice un estudio comparativo entre los filtros pasa-bajas Ideal, Gaussiano y Butterworth (con órdenes armónicas $n = 1, 2, 4$), parametrizados con frecuencias de corte $D_0 = 20, 40, 60$ píxeles. Para cada combinación estructural, calcule los índices PSNR y SSIM de la imagen resultante frente a la señal original de referencia. Organice los datos cuantitativos en una tabla estructurada y grafique las curvas unidimensionales de las funciones de transferencia correspondientes a lo largo del perfil horizontal $H(u, 0)$.

4. **(15%) Banco de Filtros Multirresolución de Haar:** Desarrolle un script para ejecutar manualmente

la descomposición *wavelet* discreta 2D de primer nivel utilizando la familia Haar. El algoritmo debe calcular los coeficientes de los filtros correspondientes pasa-bajas (h) y pasa-altas (g), aplicándolos de forma separable sobre las filas y columnas de la matriz, seguidos por la operación de diezmado (submuestreo espacial por un factor de 2). Valide numéricamente la exactitud de su implementación contrastando las subbandas obtenidas con la salida de la función `pywt.dwt2(img, 'haar')`.

5. **(15%) Compresión Dispersa por Umbralización Wavelet:** Aplique la técnica de filtrado por umbralización abrupta (*hard thresholding*) sobre los coeficientes de detalle de la descomposición *wavelet*, adoptando los umbrales numéricos $T \in \{5, 10, 20, 40, 80\}$ para las familias Haar, Daubechies (`db4`) y Symlets (`sym4`). Después de realizar el proceso de síntesis mediante la transformada inversa (`pywt.waverec2`), calcule los valores de PSNR y SSIM de cada imagen reconstruida. Identifique y justifique qué combinación de familia *wavelet* y umbral T maximiza la similitud estructural.

6. **(15%) Construcción de Codificador JPEG Simplificado:** Implemente el flujo completo de compresión de datos simulando el estándar JPEG. El flujo debe abarcar: conversión espacial $RGB \to YC_bC_r$, submuestreo cromático en la proporción 4:2:0, segmentación de la luminancia en bloques disjuntos de 8×8 píxeles, aplicación de la DCT-II 2D ortogonal y cuantización lineal basada en la matriz normalizada de luminancia escalada por factores de calidad deseados. Realice la decodificación inversa y compare cuantitativamente las reconstrucciones con los archivos generados por la función `cv2.imencode` para los factores de calidad de 20, 50 y 80.

7. **(15%) Análisis Perceptual en Contenidos Heterogéneos:** Desarrolle una imagen sintética compuesta por tres regiones distintas y de características espectrales contrastantes: una textura fotográfica compleja (representando altas frecuencias estocásticas), un área de texto vectorizado con bordes nítidos (representando transiciones escalón puras) y un gradiente lineal continuo (representando bajas frecuencias homogéneas). Someta esta imagen mixta a los procesos de compresión bajo los formatos JPEG, PNG y WebP. Evalúe e interprete los resultados correlacionando el tamaño final del archivo en disco con las métricas PSNR y SSIM obtenidas, justificando qué formato exhibe el mejor desempeño para señales de naturaleza heterogénea y por qué ocurre esa ventaja en términos de compactación de energía y preservación perceptual.

Referencias del Capítulo

La fundamentación teórica y el desarrollo analítico de los conceptos tratados en este capítulo se basan en las siguientes obras de referencia:

- **Gonzalez (2018)** — Formulaciones clásicas de Transformadas Discretas de Fourier 2D (DFT), diseño de filtros analíticos en el dominio de la frecuencia, Transformada Discreta de Cosenos (DCT) y principios fundamentales de sistemas de compresión de imágenes.
- **Oppenheim (2010)** — Teoría formal de señales y sistemas aplicados en el dominio discreto, cubriendo las propiedades matemáticas de la DFT y el modelado analítico del Teorema de la Convolución.
- **Mallat (1999)** — Fundamentación matemática de la teoría de *wavelets*, formalización del análisis multirresolución (MRA) y arquitectura de bancos de filtros diádicos.
- **Wallace (1991)** — Especificación original y aspectos de ingeniería del estándar de compresión ISO/IEC JPEG, con énfasis en los criterios psicovisuales para el diseño de matrices de cuantización DCT.
- **Szeliski (2022)** — Modelado computacional y caracterización de métricas modernas de fidelidad y calidad perceptual (PSNR y SSIM), así como el análisis comparativo de formatos de imagen rasterizados de alto rendimiento.

Executar Colab | Abrir GitHub

5.13 💻 Parte Práctica con Ejercicios de Programación

La presente lista de ejercicios de programación (EP) consolida las formulaciones teóricas presentadas a lo largo del Capítulo 5 — Transformadas y Compresión — mediante una ruta práctica aplicada. Los ejercicios se estructuran a partir de matrices de dimensiones reducidas, lo que permite la validación analítica y la inspección manual de cada coeficiente, manteniendo la consistencia metodológica adoptada en los capítulos anteriores.

El encadenamiento de los ejercicios reproduce rigurosamente el flujo conceptual del capítulo: se comienza con la implementación explícita de la Transformada Discreta de Fourier (DFT) a partir de su definición matemática fundamental; se avanza hacia el diseño de filtros pasa-baja y máscaras *notch* en el dominio de la frecuencia; se aplica la cuantización de coeficientes (núcleo de la compresión con pérdida); y se concluye con la integración de estas etapas en la construcción de un *pipeline* de compresión JPEG simplificado y en el análisis perceptual de formatos de imagen.

❗ Directrices para la Resolución de los Ejercicios de Programación

En todos los ejercicios de este capítulo, las coordenadas del **centro del espectro** (origen de las frecuencias espaciales tras la aplicación del desplazamiento `fftshift`) deben determinarse mediante división entera. Para una matriz con L filas y C columnas, la componente de frecuencia nula se localiza en la posición:

$$(c_y, c_x) = \left(\left\lfloor \frac{L}{2} \right\rfloor, \left\lfloor \frac{C}{2} \right\rfloor \right)$$

Esta convención es rigurosamente idéntica a la adoptada por la función `np.fft.fftshift`. Además, en todas las etapas que requieran discretización o redondeo numérico (ya sea en la cuantización de coeficientes AC o en la reconstrucción final de píxeles), debe emplearse el redondeo estándar al entero más cercano (*round half away from zero*), mitigando ambigüedades en valores con fracción exactamente igual a 0.5.

🎯 Objetivo de este Cuaderno

El cuaderno permite desarrollar, validar, organizar y probar soluciones de **Ejercicios de Programación (EPs)** en entornos interactivos, como Colab, con los mismos casos de prueba de Moodle, copiándolos allí solo en el momento de registrar la nota oficial.

Download

Descargue `morph.py` y `testsuite.py` ejecutando la celda a continuación:

```
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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Ejecutando las Pruebas

Para evaluar las pruebas, ejecuta `TestSuite("EP05_01.extensión").run()` en una nueva celda, reemplazando la extensión por la del lenguaje utilizado (`.py`, `.java`, `.c`, `.cpp`, `.js` o `.r`). El sistema descarga los casos de prueba de GitHub, ejecuta el programa y calcula la nota automáticamente.

Para probar código Python directamente, sin guardar archivo, usa `run_code(codigo)` pasando el código como *cadena* en una variable `codigo`:

```
codigo = """
from morph import mm
# ... tu código aquí ...
"""
TestSuite("EP05_01").run_code(codigo)
```

5.13.1 EP05_01 🟢 Filtro Pasa-Bajas Ideal por Distancia en el Espectro

En un ***escáner* de documentos antiguo**, el sensor capta papel arrugado y textura de fibra junto con el texto — ruido de alta frecuencia que "contamina" el espectro en los bordes. El técnico de mantenimiento no tiene acceso a la imagen original, solo al **espectro de magnitud ya calculado** por el software del *escáner*. Su trabajo es simple y quirúrgico: mantener únicamente el **círculo central** de bajas frecuencias (la estructura global del documento) y borrar todo lo que esté fuera del radio D_0, eliminando la textura fina sin siquiera tocar la imagen espacial.

Este es el **Filtro Pasa-Bajas Ideal (LPFI)**: la operación espectral más directa del capítulo, pero también la que mejor revela la anatomía de un espectro centrado.

5.13.1.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (filas) y C (columnas) del espectro de magnitud — ya proporcionado **centrado** (equivalente a la salida de `np.fft.fftshift`).
2. **Frecuencia de corte:** Leer el entero D_0.
3. **Datos:** Leer los valores enteros de la matriz de magnitud, fila por fila.
4. **Centro del espectro:** Calcular $(c_y, c_x) = (L\,//\,2,\ C\,//\,2)$.
5. **Distancia:** Para cada posición (u, v), calcular

$$D(u,v) = \sqrt{(u-c_y)^2 + (v-c_x)^2}$$

6. **Máscara ideal:** Aplicar

$$H(u,v) = \begin{cases} 1, & D(u,v) \le D_0 \\ 0, & D(u,v) > D_0 \end{cases}$$

7. **Filtrado:** El valor de salida es $\mathrm{mag}'(u,v) = \mathrm{mag}(u,v) \cdot H(u,v)$.
8. **Salida:** Mostrar la matriz filtrada con dimensiones $L \times C$.

5.13.1.2 📌 Restricciones Computacionales

- **Comparación no estricta:** el criterio usa $D(u,v) \le D_0$ (la frontera pertenece al filtro, es decir, se mantiene).
- **Tipo:** todos los valores de entrada y salida son enteros; la distancia se calcula en punto flotante solo internamente.
- **Sin redondeo de magnitud:** como la entrada ya es entera y la máscara es binaria (0 o 1), la salida nunca necesita redondeo.

5.13.1.3 🧠 Fundamentación Teórica

Región	Distancia al centro	Efecto del filtro
Centro ($D \leq D_0$)	Bajas frecuencias	Preservadas — estructura global mantenida
Bordes ($D > D_0$)	Altas frecuencias	Puestas a cero — textura y ruido eliminados
D_0 **pequeño**	—	Imagen reconstruida quedaría muy borrosa
D_0 **grande**	—	Poca filtración; casi toda la energía preservada

5.13.1.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Entero D_0.
- Líneas siguientes: Elementos enteros de la matriz de magnitud (centrada).

Salida:

- Matriz filtrada en L filas y C columnas, separados por espacio.

5.13.1.5 📌 Ejemplos

Entrada	Salida	Observación
3 3 1 10 20 30 40 50 60 70 80 90	0 20 0 40 50 60 0 80 0	Centro $(1, 1)$. Las esquinas tienen $D = \sqrt{2} \approx 1.41 > 1$, por lo que se ponen a cero; los vecinos ortogonales tienen $D = 1 \leq 1$ y se mantienen.
1 3 0 5 9 7	0 9 0	$L = 1, C = 3$: centro en $(0, 1)$. Solo la propia posición central ($D = 0$) sobrevive a $D_0 = 0$.

```
%%writefile EP05_01.py
# Código Python
```

```
Overwriting EP05_01.py
```

```
TestSuite("EP05_01.py").run()
```

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

5.13.2 EP05_02 🟡 Filtro *Notch*: Eliminando Picos Periódicos

Una cámara de **inspección industrial** captura imágenes de placas de circuito, pero la fuente de alimentación de la línea de producción introduce una **interferencia eléctrica periódica** — un patrón de franjas casi imperceptible a simple vista, pero que aparece en el espectro de Fourier como **pares de picos brillantes** simétricamente posicionados alrededor del centro. El equipo de visión por computadora no puede reprocesar

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap05/sim-ep0501.html

Figura 5.32: Simulador EP05_01: Filtro Pasabajas Ideal en el Espectro

la captura: necesita **localizar y borrar quirúrgicamente** esos pares de picos en el espectro, preservando todo el resto de la información útil de la imagen.

Ese es el papel del **filtro rechaza-banda *notch***: a diferencia del pasa-bajas (que afecta una región continua), ataca **puntos específicos y sus simétricos**, dejando el resto del espectro intacto.

5.13.2.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (filas) y C (columnas) del espectro de magnitud centrado.
2. **Datos:** Leer los valores enteros de la matriz de magnitud, fila por fila.
3. **Picos:** Leer el entero K (cantidad de pares de picos a eliminar).
4. **Para cada uno de los K picos:** leer tres enteros Δv, Δu, r — desplazamiento vertical, desplazamiento horizontal y radio del *notch*.
5. **Centro del espectro:** $(c_y, c_x) = (L \,//\, 2,\ C \,//\, 2)$.
6. **Supresión simétrica:** para cada pico, poner a cero **todas** las posiciones (u, v) tales que la distancia al punto $(c_y + \Delta v,\ c_x + \Delta u)$ sea $\leq r$, **y también** todas las posiciones con distancia $\leq r$ al punto simétrico $(c_y - \Delta v,\ c_x - \Delta u)$.
7. **Salida:** Mostrar la matriz resultante con dimensiones $L \times C$.

5.13.2.2 📌 Restricciones Computacionales

- **Simetría obligatoria:** cada pico informado genera **dos** discos puestos a cero (el punto y su simétrico respecto al centro) — olvidar el simétrico es el error más común.
- **Superposición:** si dos discos se superponen, la posición permanece en cero (no hay "suma" ni restauración).
- **Comparación no estricta:** una posición se pone a cero si distancia $\leq r$.
- **Orden de lectura:** los K picos deben procesarse en el orden en que aparecen en la entrada, pero el resultado final no depende del orden (las operaciones de poner a cero son conmutativas).

5.13.2.3 🧠 Fundamentación Teórica

Concepto	Papel en el filtro *notch*
Pico en $(\Delta v, \Delta u)$	Frecuencia de la interferencia periódica detectada visualmente en el espectro
Punto simétrico $(-\Delta v, -\Delta u)$	Toda DFT de señal real es hermítica: los picos siempre aparecen en pares simétricos al centro
Radio r	Controla la "anchura" del rechazo — r grande elimina más energía alrededor del pico, pero también información útil

5.13.2.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Líneas siguientes: Elementos enteros de la matriz de magnitud (centrada), L filas.
- Siguiente línea: Entero K.
- K líneas siguientes: tres enteros Δv, Δu, r (separados por espacios).

Salida:

- Matriz resultante en L filas y C columnas, separadas por espacios.

5.13.2.5 📌 Ejemplos

Entrada	Salida	Observación
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	Centro $(c_y, c_x) = (2, 2)$. El pico informado $(\Delta v, \Delta u) = (1, 1)$ genera el punto $(3, 3)$ (valor 19) y su simétrico $(1, 1)$ (valor 7), ambos puestos a cero con $r = 0$ (solo los puntos exactos).

```
%%writefile EP05_02.py
# Código Python
```

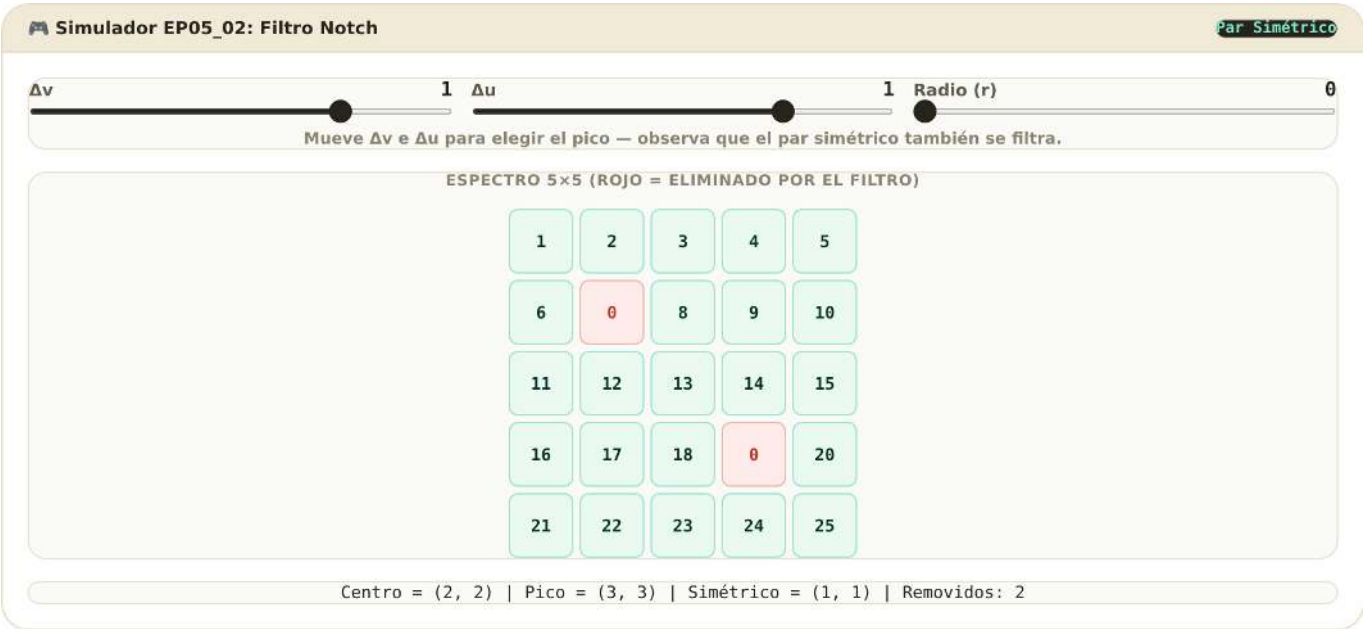
```
Overwriting EP05_02.py
```

```
TestSuite("EP05_02.py").run()
```

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

5.13.3 EP05_03 🟠 Cuantización DCT: la Verdadera Fuente de Compresión

Una aplicación de **galería de fotos** necesita reducir el tamaño de miles de imágenes antes de hacer *upload* a la nube, sin recodificar todo desde cero. El ingeniero responsable ya tiene los **coeficientes DCT** de cada bloque 4×4 calculados (la etapa costosa computacionalmente ya se ha realizado) — solo falta aplicar la **tabla de cuantización**, la etapa que realmente descarta información y genera compresión. Los coeficientes de alta frecuencia, menos perceptibles al ojo humano, reciben divisores grandes y tienden a convertirse en



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap05/sim-ep0502.html

Figura 5.33: Simulador EP05_02: Filtro Notch

cero; los coeficientes de baja frecuencia, más perceptibles, reciben divisores pequeños y sobreviven casi intactos.

Vas a implementar exactamente esta etapa: **cuantizar y descuantizar** (dividir, redondear, multiplicar de vuelta) — el corazón de la compresión *lossy* del JPEG.

5.13.3.1 📋 Directrices de Implementación

1. **Dimensión del bloque:** Leer el entero N (bloque $N \times N$).
2. **Coeficientes:** Leer la matriz C de coeficientes DCT, N filas con N enteros cada una (pueden ser negativos).
3. **Tabla de cuantización:** Leer la matriz Q, N filas con N enteros positivos cada una.
4. **Cuantización:** Para cada posición (u, v), calcular el índice cuantizado

$$\tilde{C}(u, v) = \text{round}\left(\frac{C(u, v)}{Q(u, v)}\right)$$

   usando redondeo estándar al entero más cercano (los valores intermedios `.5` nunca ocurren en los casos de prueba).
5. **Descuantización (reconstrucción):** Calcular

$$C'(u, v) = \tilde{C}(u, v) \times Q(u, v)$$

6. **Salida:** Mostrar la matriz reconstruida C', $N \times N$, enteros.

5.13.3.2 📌 Restricciones Computacionales

- ***Round-trip* completo:** la salida es el coeficiente **reconstruido** ($\tilde{C} \times Q$), no el índice cuantizado aislado.
- **División en punto flotante:** la división $C(u, v)/Q(u, v)$ debe realizarse en punto flotante antes del redondeo — la división entera truncada producirá un resultado incorrecto.
- **Signo preservado:** los coeficientes negativos mantienen el signo después de la cuantización y la reconstrucción.
- $Q(u, v) > 0$ **siempre:** no hay necesidad de tratar la división por cero.

5.13.3.3 🧠 Fundamentación Teórica

Coeficiente	Frecuencia	Valor típico de Q	Efecto de la cuantización
$C(0,0)$	DC (promedio del bloque)	Pequeño	Casi siempre sobrevive — domina la energía
$C(u,v)$ bajo $u+v$	Baja frecuencia	Pequeño/medio	Parcialmente preservado
$C(u,v)$ alto $u+v$	Alta frecuencia	Grande	Frecuentemente se convierte en cero — fuente de la compresión

5.13.3.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

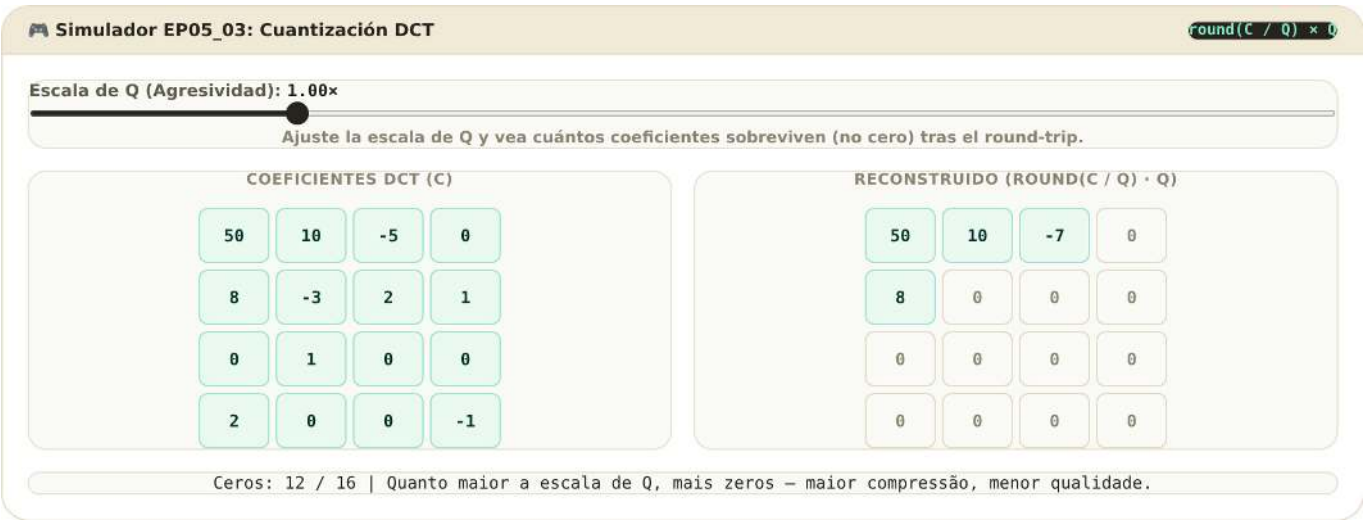
- Línea 1: Entero N.
- N líneas siguientes: matriz C (coeficientes DCT, enteros, pueden ser negativos).
- N líneas siguientes: matriz Q (tabla de cuantización, enteros positivos).

Salida:

- Matriz reconstruida C', $N \times N$, enteros separados por espacio.

5.13.3.5 📌 Ejemplos

Entrada	Salida	Observación
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	$C(0,0) = 50/2 = 25 \to 25\times 2 = 50$ (preservado). $C(0,2) = -5/7 \approx -0.71 \to -1 \to -1 \times 7 = -7$. Ya $C(1,1) = -3/7 \approx -0.43 \to 0$: anulado por la cuantización — la mayor parte del bloque se convierte en cero, ilustrando la compactación de energía en la esquina superior izquierda.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap05/sim-ep0503.html

Figura 5.34: Simulador EP05_03: Cuantización DCT (*round-trip*)

```
%%writefile EP05_03.py
# Código Python
```

```
Overwriting EP05_03.py
```

```
TestSuite("EP05_03.py").run()
```

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

5.13.4 EP05_04 🔴 Implementando la DFT 2D a partir de la definición

Un laboratorio de investigación en **astronomía computacional** recibió, de una misión antigua, un pequeño sensor experimental cuyos datos brutos no pueden ser procesados por bibliotecas modernas de FFT — el entorno de validación está aislado y solo permite operaciones aritméticas básicas. El equipo necesita **reimplementar la Transformada de Fourier Discreta 2D a partir de la propia definición matemática**, célula por célula, para después comparar bit a bit con `np.fft.fft2` en otro entorno.

Este es el ejercicio más conceptual de la lista: no hay atajos. Vas a implementar el doble sumatorio de la Ecuación 5.1 directamente, evidenciando *por qué* existe la FFT — y el costo computacional que evita.

5.13.4.1 📋 Directrices de implementación

1. **Dimensiones:** Leer los enteros M (filas) y N (columnas) de la imagen $f(x, y)$.
2. **Datos:** Leer los valores enteros de $f(x, y)$, fila a fila.
3. **DFT 2D:** Para cada par de frecuencias (u, v) con $u = 0, \dots, M-1$ y $v = 0, \dots, N-1$, calcular

$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y)\, e^{-j2\pi\left(\frac{ux}{M} + \frac{vy}{N}\right)}$$

   usando la identidad de Euler $e^{-j\theta} = \cos(\theta) - j\sin(\theta)$ para separar parte real e imaginaria — **no se debe utilizar ninguna función de FFT predefinida**.
4. **Magnitud:** Calcular $|F(u, v)| = \sqrt{\mathrm{Re}(F)^2 + \mathrm{Im}(F)^2}$ y redondear al entero más cercano.
5. **Salida:** Mostrar la matriz de magnitudes redondeadas, $M \times N$, en el mismo orden (sin `fftshift` — el DC permanece en $(0, 0)$).

5.13.4.2 📌 Restricciones computacionales

- **Prohibido usar bibliotecas de FFT:** la implementación debe calcular los sumatorios dobles explícitamente (bucles anidados), aunque sea más lenta.
- **Sin `fftshift`:** la salida mantiene la convención cruda de la DFT, con el componente DC en $F(0, 0)$ (esquina superior izquierda).
- **Redondeo:** la magnitud final debe redondearse al entero más cercano; en los casos de prueba no hay ambigüedad `.5`.
- **Precisión:** pequeños errores de punto flotante (del orden de 10^{-6}) antes del redondeo son esperados y no afectan al resultado entero final.

5.13.4.3 🧠 Fundamentación teórica

Elemento	Significado
$F(0, 0)$	Componente DC — suma de todos los píxeles, $F(0, 0) = \sum f(x, y)$
Parte real $\mathrm{Re}(F)$	Proyección de la señal sobre cosenos
Parte imaginaria $\mathrm{Im}(F)$	Proyección de la señal sobre senos

Elemento	Significado
Complejidad de esta implementación	$\mathcal{O}((MN)^2)$ — por eso la FFT, con $\mathcal{O}(MN\log(MN))$, resulta indispensable en imágenes reales

5.13.4.4 📦 Especificación de entrada y salida (VPL)

Entrada:

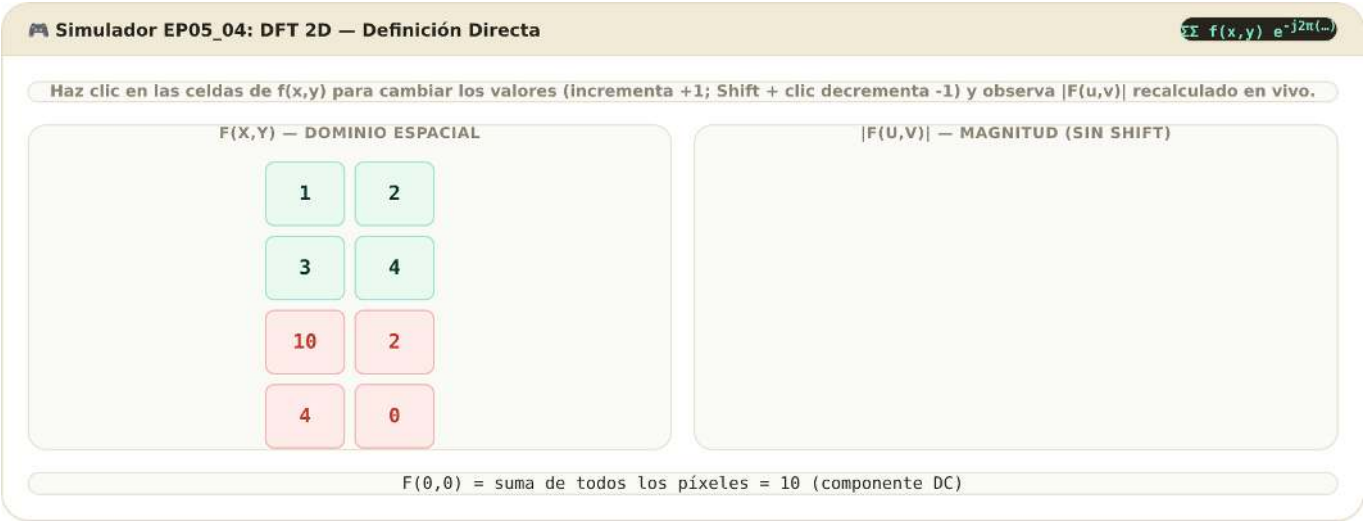
- Línea 1: Entero M.
- Línea 2: Entero N.
- Líneas siguientes: Elementos enteros de $f(x, y)$, M líneas.

Salida:

- Matriz de magnitudes $|F(u, v)|$ redondeadas, $M \times N$, separadas por espacios.

5.13.4.5 📌 Ejemplos

Entrada	Salida	Observación
2 2 1 2 3 4	10 2 4 0	$F(0,0) = 1+2+3+4 = 10$ (DC = suma total). $F(0,1) = (1-2)+(3-4) = -2 \to \lvert F\rvert = 2$. $F(1,0) = (1+2)-(3+4) = -4 \to \lvert F\rvert = 4$. $F(1,1) = (1-2)-(3-4) = 0$.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap05/sim-ep0504.html

Figura 5.35: Simulador EP05_04: DFT 2D manual

```
%%writefile EP05_04.py
# Código Python
```

```
Overwriting EP05_04.py
```

```
TestSuite("EP05_04.py").run()
```

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

5.13.5 EP05_05 🏆 *Pipeline* JPEG Completo: DCT, Cuantización y Reconstrucción

Usted ha sido contratado para crear, desde cero, un **códec JPEG didáctico** en un entorno embebido, sin ninguna biblioteca de imágenes disponible — solo operaciones matemáticas básicas. El cliente quiere entender exactamente dónde se pierde la calidad y dónde se recupera, bloque por bloque. Este es el desafío final del capítulo: integrar **todo** lo estudiado — la DCT-II ortonormal, la cuantización perceptual y la reconstrucción vía IDCT — en un único *pipeline* de extremo a extremo, procesando un bloque $N \times N$ desde el inicio hasta el final, exactamente como el estándar JPEG lo hace internamente, 8×8 píxeles a la vez.

5.13.5.1 📋 Directrices de Implementación

1. **Dimensión del bloque:** Leer el entero N.
2. **Bloque original:** Leer la matriz de píxeles $f(x, y)$, N líneas con N enteros en $[0, 255]$.
3. **Tabla de cuantización:** Leer la matriz Q, $N \times N$ enteros positivos.
4. **Centralización:** Restar 128 de cada píxel: $g(x, y) = f(x, y) - 128$.
5. **DCT-II 2D ortonormal:** Calcular

$$C(u, v) = \alpha(u)\,\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} g(x, y)\, \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right]$$

   con $\alpha(0) = \sqrt{1/N}$ y $\alpha(k) = \sqrt{2/N}$ para $k > 0$.
6. **Cuantización:** $\tilde{C}(u, v) = \text{round}(C(u, v)/Q(u, v))$.
7. **Descuantización:** $C'(u, v) = \tilde{C}(u, v) \times Q(u, v)$.
8. **IDCT-II 2D (inversa ortonormal):** Calcular $g'(x, y)$ a partir de $C'(u, v)$ usando la transformada inversa correspondiente (misma base, sumatorio sobre u, v).
9. **Reversión de la centralización y redondeo:** $f'(x, y) = \text{round}(g'(x, y) + 128)$, restringido al intervalo $[0, 255]$ (*clipping*).
10. **Salida:** Mostrar el bloque reconstruido f', $N \times N$, enteros.

5.13.5.2 📌 Restricciones Computacionales

- ***Pipeline* completo obligatorio:** todas las seis etapas (centralizar, DCT, cuantizar, descuantizar, IDCT, revertir) deben implementarse — omitir la cuantización no pasa las pruebas, pues el resultado sería idéntico al original.
- ***Clipping*:** los valores reconstruidos fuera de $[0, 255]$ deben truncarse (0 si es negativo, 255 si es mayor que 255).
- **Redondeo:** tanto en la cuantización como en la reconstrucción final de los píxeles, use redondeo estándar; los casos de prueba evitan ambigüedad `.5`.
- **Base ortonormal:** la normalización $\alpha(u)$ y $\alpha(v)$ debe aplicarse exactamente como se especifica — sin ella, la IDCT no reconstruye correctamente.

5.13.5.3 🧠 Fundamentación Teórica

Etapa	Análoga en el estándar JPEG real	Dónde se pierde la calidad
Centralización	Misma — la DCT asume señal centrada en cero	Sin pérdida
DCT-II	Etapas 3–4 del *pipeline* (Tabla 5.7)	Sin pérdida (transformación exacta y reversible)
Cuantización	Etapa 5 — división por $Q(u, v)$	**Principal fuente de pérdida** — los coeficientes de alta frecuencia se vuelven cero
IDCT	Reconstrucción final	Reconstruye exactamente los coeficientes *cuantizados*, no los originales

5.13.5.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero N.
- N líneas siguientes: bloque original $f(x, y)$, enteros en $[0, 255]$.
- N líneas siguientes: tabla de cuantización Q, enteros positivos.

Salida:

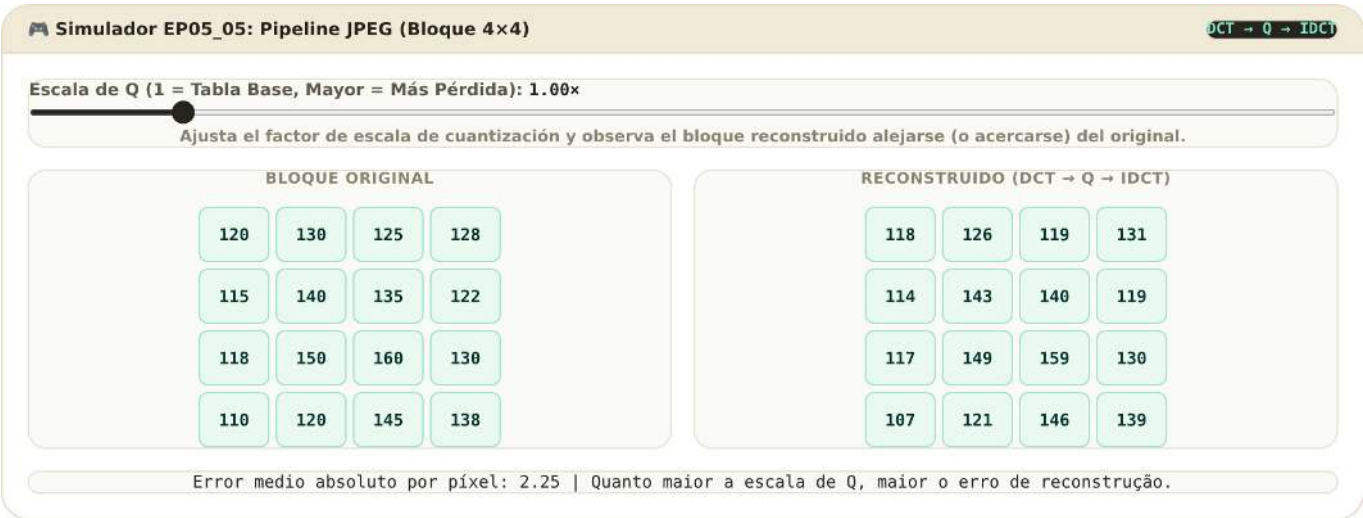
- Bloque reconstruido $f'(x, y)$, $N \times N$, enteros en $[0, 255]$, separados por espacios.

5.13.5.5 📌 Ejemplos

Entrada	Salida	Observación
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	Tras la DCT, cuantización agresiva en las altas frecuencias (valores grandes de Q en la esquina inferior derecha) y reconstrucción vía IDCT, el bloque queda **cercano** al original, pero no idéntico — la diferencia es el costo de la compresión *lossy*.

5.13.5.6 💡 Consejo de Depuración

Si el resultado no coincide, verifique en este orden: (1) los coeficientes DCT brutos (antes de la cuantización) — deben reconstruir el original **exactamente** vía IDCT si omite las etapas 6–7; (2) la tabla $\alpha(u)$ — error común es aplicar $\sqrt{2/N}$ también para $u = 0$; (3) el redondeo de la cuantización, que debe ocurrir **antes** de multiplicar de vuelta por Q.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap05/sim-ep0505.html

Figura 5.36: Simulador EP05_05: *Pipeline* JPEG completo en bloques

```
%%writefile EP05_05.py
# Código Python
```

```
Overwriting EP05_05.py
```

```
TestSuite("EP05_05.py").run()
```

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

Parte II

Parte II — Visión por Computador

Capítulo 6

Análisis de Documentos e Inspección Industrial

Executar Colab Abrir GitHub

En la **Parte I — Procesamiento Digital de Imágenes (PDI)**, se estudiaron técnicas para la transformación y mejora de imágenes, como operaciones morfológicas, filtrado espacial, convoluciones, umbralización, segmentación y procesamiento en el dominio de la frecuencia.

La **Parte II — Visión por Computador (VC)** amplía este alcance al abordar la interpretación automática del contenido visual, involucrando la extracción de información, el reconocimiento de patrones y la toma de decisiones a partir de imágenes.

Este capítulo presenta esa transición mediante dos aplicaciones representativas:

1. **Análisis Automatizado de Documentos**, aplicado al procesamiento de formularios, evaluaciones y otros documentos estructurados mediante sistemas de reconocimiento óptico de marcas (*Optical Mark Recognition* – OMR);
2. **Inspección Industrial Automatizada**, orientada al control de calidad y a la detección de defectos en líneas de producción.

Estas aplicaciones integran técnicas de detección de estructuras geométricas, extracción de descriptores invariantes, reconocimiento de patrones y clasificación de objetos, constituyendo la base de diversos sistemas modernos de inspección visual y automatización.

6.1 Objetivos del Capítulo

Al final de este capítulo, el estudiante deberá ser capaz de:

- **Evaluar la influencia del preprocesamiento** en la calidad del reconocimiento automático de información en documentos;
- **Realizar reconocimiento óptico de caracteres** (OCR) para convertir documentos escaneados en texto codificado;
- **Aplicar técnicas de procesamiento de lenguaje natural**, incluida la traducción automática, al texto obtenido por OCR;
- Aplicar técnicas de **alineación y rectificación geométrica de documentos** utilizando la Transformada de Hough y transformaciones proyectivas;
- **Implementar sistemas de reconocimiento óptico de marcas** (OMR) para la lectura automatizada de evaluaciones y formularios;
- **Detectar y segmentar regiones de interés** con base en operaciones morfológicas, contornos y propiedades geométricas;
- **Decodificar marcadores bidimensionales y códigos de barras**, integrando bibliotecas de Visión por Computador a *pipelines* de procesamiento documental;
- **Desarrollar *pipelines* de Visión por Computador** para el análisis automatizado de documentos.

Este capítulo marca la transición del **Procesamiento Digital de Imágenes**, orientado a la transformación de imágenes, hacia la **Visión por Computador**, cuyo objetivo es interpretar el contenido visual, extraer información y respaldar procesos automatizados de análisis y toma de decisiones.

6.2 Configuración del Entorno

Los ejemplos de este capítulo utilizan bibliotecas ampliamente empleadas en PDI-VC. El bloque siguiente instala los paquetes necesarios; en entornos que ya los posean, la ejecución puede omitirse.

```
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

# instalar más dependencias además de morph.py para este capítulo
import sys, subprocess, importlib, shutil

def setup_cap06():
    """Instala dependencias de sistema y Python específicas del Capítulo 6
    (OCR, lectura de PDF, código de barras)."""

    # 1. Dependencias de sistema
    if 'google.colab' in sys.modules:
        print("[AMBIENTE] Google Colab. Configurando dependencias del sistema...")
        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:
        # Ambiente local sin tesseract: intenta instalar vía apt-get (requiere sudo/root)
        if shutil.which("apt-get"):
            print("[AMBIENTE] Local. Instalando tesseract-ocr vía apt-get (puede pedir
            contraseña)...")
            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(
                    "[AVISO] No fue posible instalar automáticamente. "
                    "Instale manualmente: sudo apt install tesseract-ocr tesseract-ocr-por"
                )
        else:
            print(
                "[AVISO] tesseract no encontrado y apt-get no disponible. "
                "Instale manualmente antes de ejecutar las celdas de OCR."
            )

    # 2. Dependencias del Python (instala solo las ausentes)
    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"[AVISO] Fallo al instalar {pkg} (necesario para el módulo {mod}).")


setup_cap06()

# 3. Imports globales del pipeline
import cv2, numpy as np, matplotlib.pyplot as plt
from skimage import io, data, color
```

```
 Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0
[AMBIENTE] Local. Instalando tesseract-ocr vía apt-get (puede pedir contraseña)...
[AVISO] No fue posible instalar automáticamente. Instale manualmente: sudo apt install
tesseract-ocr tesseract-ocr-por
```

Además de estas bibliotecas, se utilizará el módulo didáctico `morph.py`, desarrollado para simplificar operaciones de lectura, visualización y procesamiento de imágenes a lo largo de este libro. El siguiente código verifica su disponibilidad, realiza la *descarga* cuando sea necesario y confirma la versión cargada.

```
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"  Ambiente listo. morph {getattr(morph, '__version__', 'local_file')}")
```

```
 Ambiente listo. morph 1.1.9
```

6.3 Bases de Imágenes para Experimentación

Los ejemplos presentados en esta parte del libro utilizan, siempre que sea posible, documentos digitalizados, hojas de respuestas, códigos de barras, *QRCodes* y otras imágenes provenientes de aplicaciones reales. Para hacer que los experimentos sean totalmente reproducibles — incluso en entornos sin acceso a internet o a los archivos originales del MCTest —, también se emplean imágenes públicas ampliamente utilizadas en la enseñanza y la investigación en Procesamiento Digital de Imágenes y Visión por Computadora (PDI-VC).

¿Por qué utilizar un banco de imágenes de *benchmark*?

Imágenes como `camera()` y `coins()` se han utilizado durante décadas en libros de texto, artículos científicos y materiales didácticos de PDI-VC. Su uso ofrece ventajas importantes:

- **Reproducibilidad:** cualquier lector obtiene exactamente las mismas imágenes, independientemente del ordenador o sistema operativo utilizado, sin necesidad de *descargas* externas ni de archivos específicos de este libro;
- **Comparabilidad:** los resultados producidos pueden compararse directamente con los reportados

en la literatura, ya que los mismos conjuntos de imágenes se adoptan ampliamente como referencia;
- **Enfoque en los algoritmos:** por ser imágenes compactas, bien documentadas y distribuidas sin restricciones de uso con fines educativos y científicos, permiten concentrar la atención en las técnicas de procesamiento, reduciendo interferencias relacionadas con la adquisición o la gestión de los datos.

6.3.1 Imágenes Públicas con `skimage.data`

El módulo `skimage.data` pone a disposición una colección de imágenes de referencia ampliamente utilizada en actividades de enseñanza, investigación y validación de algoritmos en PDI-VC. La inspección del atributo `data.__all__` muestra que la versión actual reúne **42 ítems**, incluyendo fotografías naturales, documentos digitalizados, imágenes médicas, microscopía, texturas, patrones sintéticos, modelos tridimensionales y secuencias temporales. Conviene observar que algunos de estos ítems corresponden a funciones utilitarias, como `data_dir()` y `download_all()`, y no a imágenes propiamente dichas.

Como el objetivo de este capítulo es presentar aplicaciones de análisis documental e inspección visual, la Tabla 6.2 reúne una **selección representativa** de las imágenes más relevantes, organizada de acuerdo con sus principales aplicaciones en Visión Computacional. La Figura 6.1 presenta una muestra de estas imágenes, agrupadas conforme a la misma clasificación adoptada en la tabla.

Imágenes utilizadas en este capítulo

Aunque el `skimage.data` disponga de decenas de imágenes de referencia, solo cuatro son empleadas directamente en los experimentos de este capítulo. Fueron seleccionadas por reproducir, de forma controlada, características frecuentemente encontradas en documentos digitalizados y en sistemas de inspección visual industrial. La Tabla 6.1 resume el papel de cada una de ellas a lo largo de este capítulo.

Tabla 6.1: Imágenes de skimage.data utilizadas en los experimentos de este capítulo.

Imagen	Aplicación en el capítulo
`data.page()`	Página digitalizada utilizada en los experimentos de corrección de iluminación, realce local (CLAHE) y umbralización automática por Otsu.
`data.text()`	Documento que contiene texto impreso, empleado para ilustrar segmentación, extracción de contornos y etapas típicas de OCR y OMR.
`data.coffee()`	Fotografía en color con variaciones naturales de iluminación, utilizada para ejemplificar técnicas aplicables a escenas reales no documentales.
`data.brick()`	Textura de referencia empleada en ejemplos de inspección superficial y detección de defectos por análisis de varianza local.

Tabla 6.2: Selección de imágenes públicas representativas disponibles en el módulo *skimage.data*, organizadas por área de aplicación.

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*.

Categoria	Função	Descrição
🔵 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.
🔵 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

# Imágenes ordenadas por categoría; el color del título reproduce el color de la categoría en
la tab. anterior
imgs = [
    ("page",         data.page(),         "#2563eb"),   # Documentos y OCR/OMR
    ("text",         data.text(),         "#2563eb"),
    ("coins",        data.coins(),        "#16a34a"),   # Segmentación y morfología
    ("binary_blobs", data.binary_blobs(), "#16a34a"),
    ("brick",        data.brick(),        "#ea580c"),   # Textura e inspección industrial
    ("checkerboard", data.checkerboard(),"#ea580c"),
    ("camera",       data.camera(),       "#7c3aed"),   # Fotografías clásicas de PDI/VC
    ("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()
```

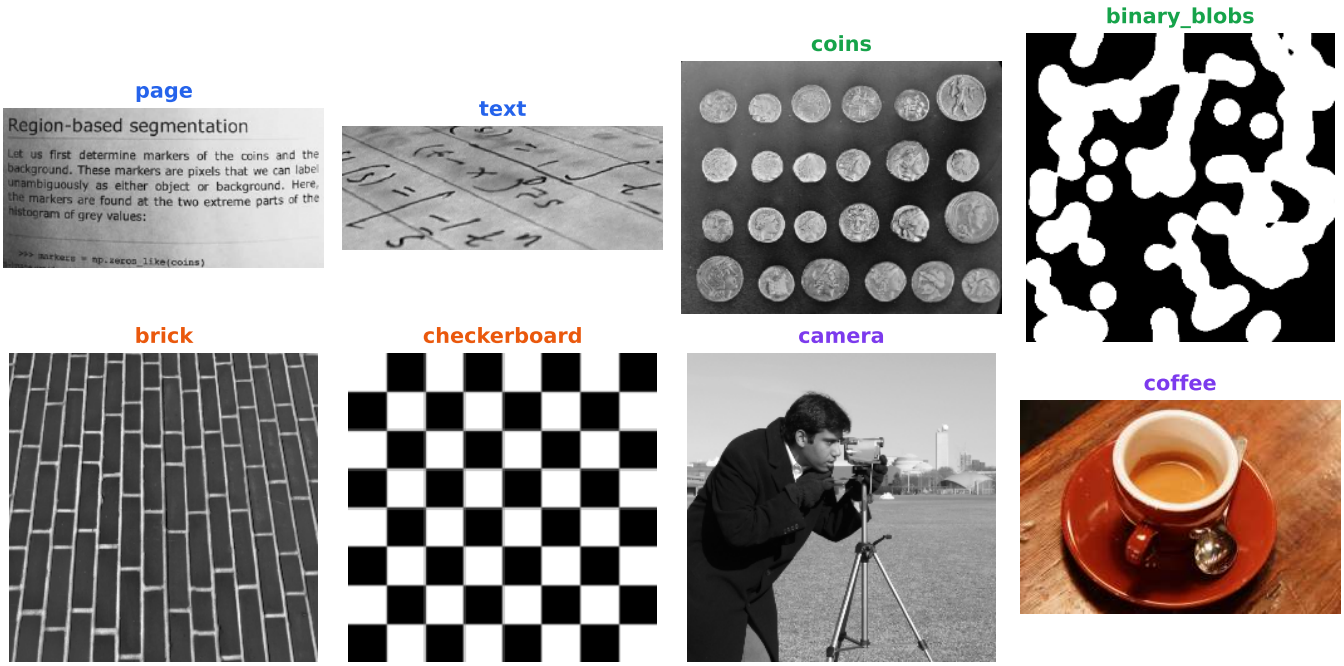


Figura 6.1: Muestra de imágenes públicas de *skimage.data*, agrupadas por área de aplicación.

6.4 Normalización de Fondo y Ecualización Local de Contraste

La calidad de la segmentación depende directamente de las características de la imagen de entrada. En documentos digitalizados, las variaciones de iluminación, las sombras, las regiones sobreexpuestas y las diferencias de tonalidad del papel reducen el contraste entre el primer plano y el fondo, dificultando la aplicación de métodos de umbralización global, como el algoritmo de Otsu.

Para minimizar estos efectos, se emplean dos técnicas complementarias de preprocesamiento:

- **Normalización de fondo:** estima la componente de baja frecuencia de la imagen mediante un fuerte suavizado y, a continuación, normaliza la imagen original con respecto a ese fondo estimado. Este procedimiento reduce los gradientes de iluminación y compensa las variaciones lentas de intensidad, preservando las estructuras de interés.
- **CLAHE** (*Contrast Limited Adaptive Histogram Equalization*), presentado en el **Capítulo 4:** divide la imagen en pequeñas regiones (*tiles*) y realiza la ecualización del histograma de cada región de forma independiente. El contraste está limitado para evitar la amplificación excesiva del ruido, lo que hace que la técnica sea especialmente adecuada para imágenes con variaciones locales de iluminación.

La Figura 6.2 compara estas estrategias utilizando la imagen `page()` de la biblioteca `skimage.data`. Se presentan seis resultados: (a) la imagen original; (b) la binarización directa mediante el método de Otsu, utilizada como referencia; (c) el fondo estimado por filtrado gaussiano; (d) la imagen después de la normalización de fondo; (e) la binarización obtenida tras la aplicación del CLAHE seguida del método de Otsu; y (f) la binarización obtenida tras la normalización de fondo seguida de la aplicación del método de Otsu.

La comparación permite observar el efecto producido por cada etapa del preprocesamiento y su influencia en la calidad de la segmentación. En particular, la normalización de fondo reduce las variaciones globales de iluminación, mientras que el CLAHE aumenta el contraste local entre caracteres y fondo. Dependiendo de las características de la imagen, una u otra estrategia puede producir resultados superiores, no existiendo una técnica universalmente más adecuada.

```
import cv2
from skimage import data
```

```python
from morph import mm

img = data.page()  # o: img = mm.gray(img_final) - con imagen de la hoja de prueba

#   Método 1: Normalización de fondo + Otsu
# Estima el fondo con un filtro Gaussiano de sigma grande (variaciones lentas de luz)
# y divide píxel a píxel para cancelar el gradiente de iluminación
bg = cv2.GaussianBlur(img, (0, 0), sigmaX=25)
img_norm = cv2.divide(img, bg, scale=255)
img_norm_otsu = mm.threshold(img_norm)

#   Método 2: CLAHE + Otsu
# tileGridSize define el tamaño de cada región local (tile)
# clipLimit controla el techo de amplificación - valores altos aumentan contraste
# pero también amplifican ruido
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
img_clahe = clahe.apply(img)
img_clahe_otsu = mm.threshold(img_clahe)

#   Método 0: Otsu directo (sin preprocesamiento) - referencia
img_otsu = mm.threshold(img)

mm.show(
    [img,  img_otsu, bg, img_norm, img_clahe_otsu, img_norm_otsu],
    titles=["(a) Original", "(b) Otsu directo", "(c) Gaussiano",  \
            "(d) Fondo normalizado (img/bg)", "(e) CLAHE + Otsu", \
            "(f) Normaliz. fondo + Otsu"],
    cols=3,
    figsize=(14, 8)
)
```

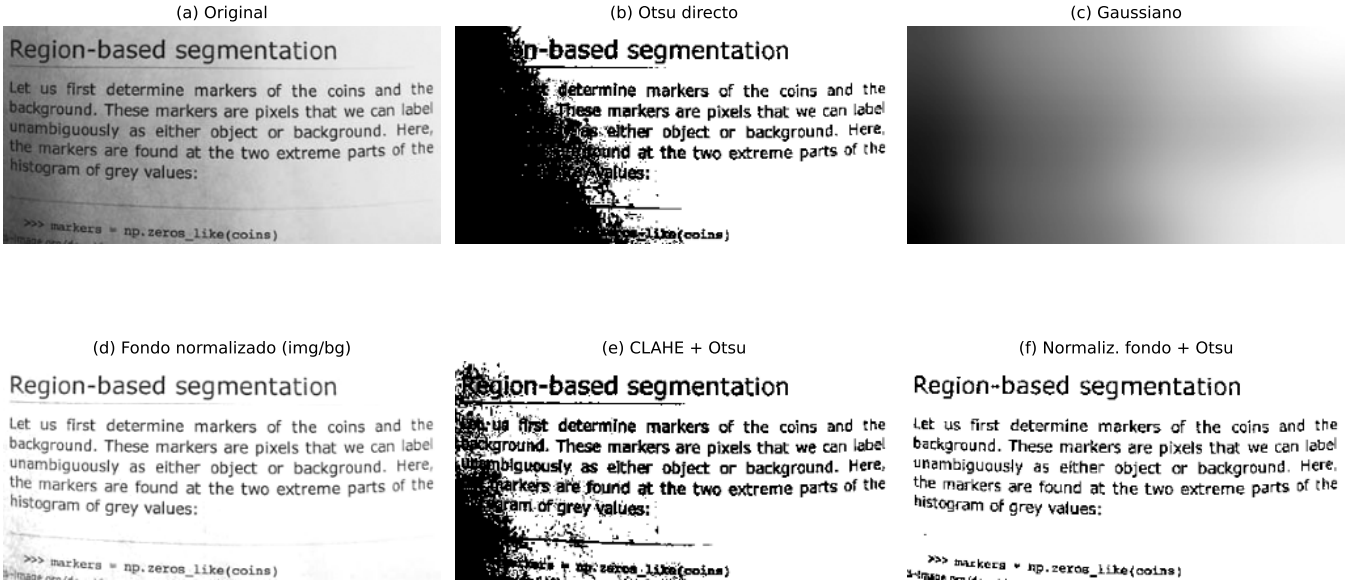


Figura 6.2: Comparación entre estrategias de preprocesamiento para binarización de la imagen de texto: (a) imagen original; (b) umbralización directa por el método de Otsu; (c) fondo estimado por filtrado Gaussiano; (d) imagen tras normalización de fondo (división por la imagen suavizada); (e) CLAHE seguido de umbralización por Otsu; y (f) normalización de fondo seguida de umbralización por Otsu. Las imágenes ilustran el efecto de cada técnica en la compensación de variaciones de iluminación y en la calidad de la segmentación.

¿Por qué funciona? — Normalización de fondo vs. CLAHE

Normalización de fondo: al dividir la imagen por una versión fuertemente suavizada de sí misma, se eliminan las variaciones lentas de luminosidad (gradiente de luz, sombra de borde) sin afectar los detalles finos — texto, líneas, burbujas. El resultado es una imagen con iluminación aproximadamente uniforme, donde el umbral global de Otsu pasa a funcionar bien en toda la página.
CLAHE: un histograma global ecualizado "estira" los tonos de toda la imagen de una vez — útil cuando la iluminación es uniforme, pero problemático cuando no lo es. El CLAHE divide la imagen en pequeños bloques (*tiles*) y ecualiza cada uno por separado, con un límite máximo de amplificación (`clipLimit`) para no hacer explotar el ruido. Es especialmente eficaz para resaltar regiones subexpuestas localmente, pero no elimina gradientes globales — por eso, aplicarlo tras la normalización de fondo tiende a producir resultados más consistentes.

6.5 Reconocimiento Óptico de Caracteres (OCR)

Tras la binarización, la siguiente etapa del procesamiento documental consiste en la conversión de la representación visual de los caracteres en texto codificado digitalmente, proceso denominado **Reconocimiento Óptico de Caracteres** (*Optical Character Recognition* — **OCR**).

De forma general, un sistema de OCR comprende tres etapas:

1. **Segmentación:** identifica líneas, palabras y caracteres en la imagen, utilizando proyecciones horizontales y verticales o detección de componentes conexos.
2. **Extracción de características:** representa cada carácter mediante atributos visuales, como bordes, curvaturas y patrones de trazo.
3. **Reconocimiento:** asocia los atributos extraídos al carácter más probable. Los sistemas actuales utilizan predominantemente redes neuronales recurrentes (LSTM) o arquitecturas basadas en *transformers*.

En este libro, se utiliza el **Tesseract OCR**, accesible mediante la biblioteca `pytesseract` (instalación: `pip install pytesseract`). Desarrollado originalmente por Hewlett-Packard entre 1985 y 1995 y actualmente mantenido por Google, Tesseract se describe en Smith (2007) y Smith (2013).. En las versiones recientes, el reconocimiento textual se realiza mediante redes neuronales LSTM.

El rendimiento del OCR depende de la calidad de la imagen de entrada. El ruido, el bajo contraste, las distorsiones geométricas y la iluminación irregular reducen la tasa de reconocimiento. Por este motivo, etapas como la rectificación, la normalización del fondo y la ecualización adaptativa (CLAHE) integran el preprocesamiento de la imagen.

¿El Tesseract necesita una imagen binarizada?

El Tesseract incorpora internamente una etapa de binarización adaptativa antes del reconocimiento de los caracteres. Por este motivo, proporcionar al OCR una imagen previamente binarizada no siempre produce los mejores resultados.
Como la umbralización es una operación irreversible, puede eliminar variaciones sutiles de intensidad en los bordes de los caracteres, como el *antialiasing*, que pueden ayudar al mecanismo de reconocimiento. En muchos casos, una imagen en tonos de gris, con buena iluminación y contraste, produce una transcripción más fiel que su versión binarizada.

Para investigar este efecto, se compara el texto extraído por Tesseract a partir de cuatro versiones de la misma imagen, presentadas en la Figura 6.3: (a) imagen original; (b) imagen sometida a la ecualización adaptativa (CLAHE) seguida de la umbralización mediante el método de Otsu; (c) imagen sometida a la normalización del fondo seguida de la umbralización mediante el método de Otsu; y (d) imagen sometida únicamente a la normalización del fondo, preservando los tonos de gris.

La comparación entre las versiones (c) y (d) muestra que, en fragmentos que contienen caracteres visualmente

similares, la versión en tonos de gris (d) produjo una transcripción más fiel al texto original que la versión binarizada (c). Este resultado indica que la umbralización aplicada en el preprocesamiento puede eliminar información útil para el reconocimiento. Así, aunque la binarización sea esencial para diversas operaciones de procesamiento de imágenes, no constituye necesariamente la mejor entrada para el OCR. La elección de la técnica de preprocesamiento debe considerar la etapa subsiguiente del *pipeline* documental.

```
import pytesseract
import shutil as _sh
if _sh.which("tesseract") is None:
    # Ambiente de build sin Tesseract instalado (ej.: sin apt/sudo):
    # degrada en lugar de romper el render. En Colab/local con Tesseract,
    # nada cambia.
    _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()

#   Reaprovechando los resultados de la sección anterior
img_otsu = mm.threshold(img)

bg = cv2.GaussianBlur(img, (0, 0), sigmaX=25)
img_norm = cv2.divide(img, bg, scale=255)       # tonos de gris, sin Otsu
img_norm_otsu = mm.threshold(img_norm)          # binarizada

clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
img_clahe = clahe.apply(img)
img_clahe_otsu = mm.threshold(img_clahe)

#   Configuración de Tesseract
# --psm 6: asume un único bloque uniforme de texto (adecuado para la imagen `page`)
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) Normaliz. fondo + Otsu",
            "(d) Normaliz. fondo (gris)"],
    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]
```

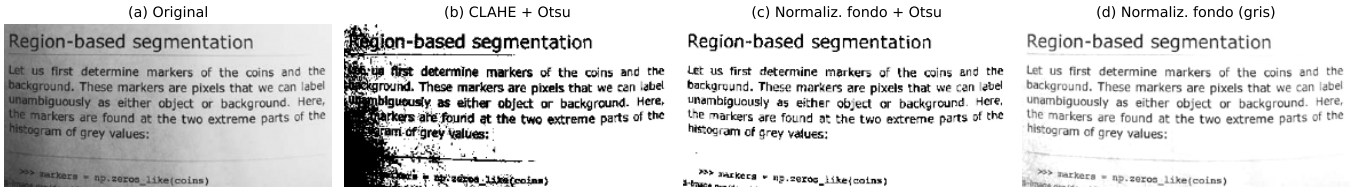


Figura 6.3: Comparación del texto extraído por Tesseract a partir de cuatro versiones de la misma imagen: (a) imagen original; (b) CLAHE seguido de umbralización por Otsu; (c) normalización de fondo seguida de umbralización por Otsu; y (d) normalización de fondo en tonos de gris, sin umbralización. La comparación evidencia que la binarización externa no siempre favorece el reconocimiento, ya que Tesseract ya realiza su propia binarización adaptativa internamente.

¿Por qué el preprocesamiento mejora el OCR?

El rendimiento del OCR depende directamente de la calidad de la imagen de entrada. El bajo contraste, la iluminación no uniforme, el ruido y las distorsiones geométricas dificultan la separación entre texto y fondo y aumentan la probabilidad de errores de reconocimiento.
Técnicas como la normalización de fondo y la ecualización adaptativa (CLAHE) corrigen gradientes de iluminación y realzan el contraste local entre los caracteres y el fondo, produciendo imágenes más adecuadas para el reconocimiento automático. La umbralización, por su parte, debe aplicarse con cautela: al tratarse de una operación irreversible, puede eliminar variaciones sutiles de intensidad en los bordes de los caracteres — como el *antialiasing* — que el propio motor de OCR utiliza internamente para resolver ambigüedades entre símbolos visualmente similares. Por este motivo, las imágenes en tonos de gris, corregidas únicamente en cuanto a la iluminación, frecuentemente producen transcripciones más fieles que sus versiones binarizadas.
En documentos capturados por cámaras de dispositivos móviles, el preprocesamiento tiende a proporcionar una mayor ganancia de rendimiento que en documentos digitalizados por *scanner*, en los cuales la iluminación suele ser más uniforme.

6.6 Traducción Automática del Texto Reconocido

Tras el reconocimiento óptico de caracteres, el texto obtenido puede someterse a técnicas de procesamiento de lenguaje natural, como corrección ortográfica, indexación, resumen y traducción automática.

La traducción automática constituye una etapa independiente del OCR. Mientras que el OCR convierte los caracteres presentes en la imagen en texto codificado, la traducción opera sobre ese texto en el idioma original del documento. De esta forma, los errores de reconocimiento pueden propagarse a la traducción, comprometiendo la calidad del resultado. Los sistemas actuales de traducción automática utilizan predominantemente arquitecturas neuronales basadas en mecanismos de atención y *transformers* [Bahdanau; Cho; Bengio (2015); Vaswani (2017)].

En este ejemplo, se utiliza el texto obtenido a partir de la imagen sometida únicamente a la normalización de fondo, sin umbralización (ítem **d** de la Figura 6.3), por presentar la transcripción más fiel entre las estrategias evaluadas en la sección anterior.

La traducción se realiza mediante la biblioteca `deep-translator` (instalación: `pip install deep-`

`translator`), que proporciona una interfaz para diferentes servicios de traducción automática, incluido Google Translate.

```
from deep_translator import GoogleTranslator
import shutil as _sh

# Texto obtenido por el OCR a partir de la imagen con normalización de fondo (tonos de gris)
texto_en = texto_norm_gray

if _sh.which("tesseract") is None:
    # Ambiente de build sin Tesseract instalado (ver celda anterior): texto_en
    # ya es el placeholder de OCR no disponible, no texto real - omitir la traducción
    # en lugar de fallar al intentar traducir una cadena que no es inglés real.
    texto_pt = "[Tradução indisponível neste ambiente - Tesseract OCR ausente]"
else:
    texto_pt = GoogleTranslator(source="en", target="pt").translate(texto_en)

print("--- Texto original generado por la imagen normalizada en tonos de gris (OCR, EN) ---")
print(texto_en)

print("\n--- Texto traducido (ES) ---")
print(texto_pt)

mm.show(
    [img_norm],
    titles=["Imagen con normalización de fondo (tonos de gris)"],
    cols=1,
    figsize=(6, 4)
)
```

```
--- Texto original generado por la imagen normalizada en tonos de gris (OCR, EN) ---
[Tesseract OCR indisponivel neste ambiente - texto omitido]

--- Texto traducido (ES) ---
[Tradução indisponível neste ambiente - Tesseract OCR ausente]
```

Imagen con normalización de fondo (tonos de gris)

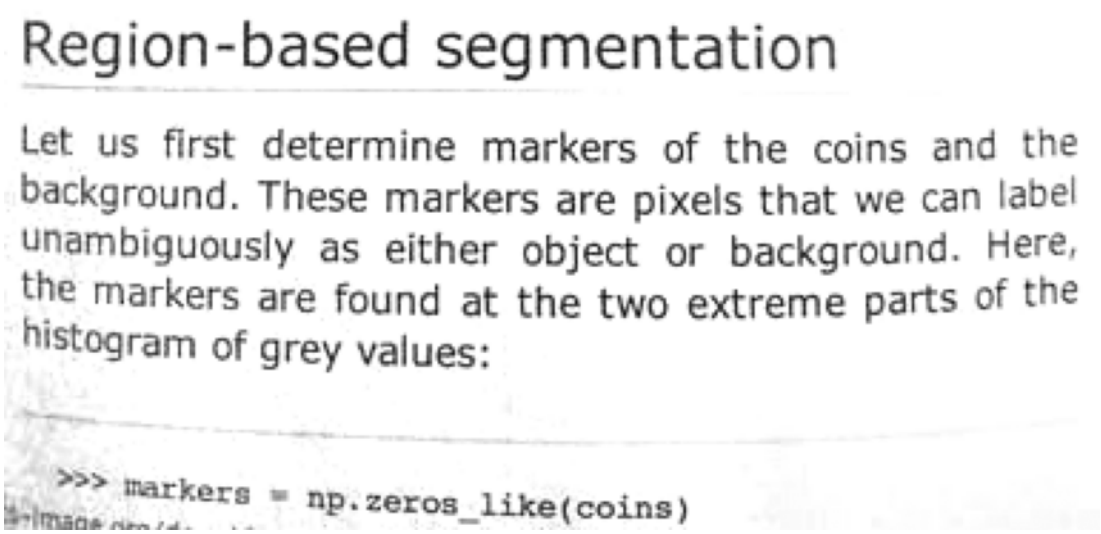


Figura 6.4: Flujo de reconocimiento y traducción automática. La imagen preprocesada por normalización de fondo, sin umbralización, se utiliza como entrada para Tesseract OCR, y el texto reconocido se traduce del inglés al español mediante la biblioteca *deep-translator*.

¿Por qué la calidad del OCR influye en la traducción?

La traducción automática utiliza como entrada el texto producido por el OCR. Los errores de reconocimiento, como caracteres incorrectos, palabras incompletas o fragmentadas, se propagan a la etapa de traducción y pueden alterar el significado del texto.
En consecuencia, la calidad de la traducción depende directamente de la fidelidad de la transcripción obtenida por el OCR. Como se discutió anteriormente, esta fidelidad no siempre se maximiza mediante una binarización externa: las imágenes en tonos de gris, corregidas solo en cuanto a la iluminación, pueden preservar información relevante para la distinción entre caracteres visualmente similares. Así, el preprocesamiento de la imagen —y la elección adecuada de sus etapas en función de la tarea posterior— contribuye a mejorar no solo el reconocimiento de los caracteres, sino también el rendimiento de etapas posteriores de procesamiento de lenguaje natural, como la traducción, la indexación y la sumarización.

6.7 Fundamentos de OMR e Inspección Industrial

El **Reconocimiento Óptico de Marcas** (*Optical Mark Recognition* — OMR) es una técnica de Visión por Computadora destinada a la identificación automática de marcaciones en posiciones previamente definidas de un formulario. Sus aplicaciones incluyen hojas de respuestas, cuestionarios, formularios administrativos y otros documentos estructurados.

A diferencia del OCR (*Optical Character Recognition*), que reconoce caracteres y palabras, el OMR determina la presencia, la ausencia o la intensidad de marcas en regiones previamente conocidas. En lugar de interpretar texto, explora propiedades geométricas y estadísticas asociadas al llenado de dichas regiones.

Los sistemas modernos de OMR procesan imágenes obtenidas mediante *escáneres*, cámaras o dispositivos móviles, automatizando tareas que anteriormente dependían de equipos especializados.

De manera general, un sistema de OMR comprende las siguientes etapas:

1. **Adquisición:** conversión del documento físico a formato digital;
2. **Preprocesamiento:** corrección geométrica, reducción de ruidos y binarización;
3. **Localización de las regiones de interés:** identificación de las áreas destinadas a las marcaciones;
4. **Análisis de las marcaciones:** evaluación del llenado de las regiones candidatas;
5. **Interpretación:** conversión de las marcaciones en respuestas o datos estructurados.

Estos principios se extienden naturalmente a la **Inspección Industrial Automatizada**. En líneas de producción, el mismo encadenamiento — adquisición, preprocesamiento, segmentación, extracción de características y decisión — se emplea para detectar defectos superficiales, verificar la integridad de componentes y medir dimensiones con precisión subpíxel. La diferencia reside en el dominio de aplicación: mientras que el OMR opera sobre documentos con estructura predefinida, la inspección industrial lidia con objetos cuyas variaciones geométricas y radiométricas deben modelarse de forma más flexible.

En las secciones siguientes, ambas aplicaciones se desarrollan mediante proyectos prácticos que reproducen etapas típicas de sistemas reales.

6.8 Proyectos Prácticos: Construcción de un *Pipeline* de Análisis Documental

Los conceptos de este capítulo se desarrollarán mediante proyectos que reproducen etapas típicas de sistemas reales de análisis documental, introduciendo técnicas reutilizables en aplicaciones de OCR, OMR, inspección visual y procesamiento de formularios.

6.8.1 Alineación Automática de Documentos (*OCR/OMR Pre-processing*)

La corrección de inclinación (*deskew*) es una etapa fundamental en el procesamiento de documentos. Las rotaciones introducidas durante la digitalización o captura comprometen la localización de regiones de interés y reducen la precisión de las etapas subsiguientes.

En este proyecto se desarrollará un sistema para estimar automáticamente la orientación predominante del documento y corregir su inclinación. Para ello, se emplearán técnicas clásicas de detección de bordes con el operador de Canny y detección de líneas mediante la Transformada de Hough. A partir de las líneas identificadas, se estimará el ángulo de rotación y se aplicará una transformación afín para producir una versión alineada del documento.

Como los formularios y las hojas de respuestas se distribuyen frecuentemente en formato PDF, el *pipeline* comienza con la rasterización de cada página, convirtiéndola en una imagen matricial. En este capítulo, dicha etapa se realizará con la biblioteca `pdf2image`, generando imágenes PNG con una resolución de 300 DPI (*dots per inch*). A partir de ellas, se podrán aplicar las técnicas de detección de bordes, Transformada de Hough, segmentación, extracción de contornos y reconocimiento automático de patrones estudiadas a lo largo del capítulo.

```
import os
import urllib.request
from pdf2image import convert_from_path
from skimage import data
import cv2

# Directorio de los microdatos y hojas de respuestas del examen institucional
file_path = "dados/provas_qrcode_EP.pdf"
url_github = (
    "https://raw.githubusercontent.com/fzampirolli/"
    "pdi-vc/master/all/cap06/dados/provas_qrcode_EP.pdf"
)

# Si el archivo no existe localmente, se descarga automáticamente desde GitHub
if not os.path.exists(file_path):
    print(f"[DOWNLOAD] Descargando PDF desde GitHub: {url_github}")
    try:
        # Garantiza que la carpeta 'datos' exista antes de guardar
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
        urllib.request.urlretrieve(url_github, file_path)
        print("[DOWNLOAD] ¡PDF descargado con éxito!")
    except Exception as e:
        print(f"[DOWNLOAD] Error al descargar el archivo: {e}")

print(f"PDF de hojas de examen digitalizadas: {file_path}")

if os.path.exists(file_path):
    # Rasterización de las páginas con resolución optimizada de 300 DPI
    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"[INGESTA] Página PDF convertida con éxito: {saida}")
else:
    print("[AVISO] Archivo PDF no localizado en la ruta. Activando fallback mediante
    skimage.data.")
    # Inyecta matriz de texto pública para garantizar la ejecución continua del pipeline
    img_fallback = data.text()
    cv2.imwrite("test02.png", img_fallback)
    print("[INGESTA] Imagen de fallback estructurada: test02.png")
```

```python
# Carga y muestra la imagen rasterizada inicial utilizando el ecosistema morph
if os.path.exists('test02.png'):
    img_original = mm.read('test02.png')
else:
    # Fallback definitivo en caso de que incluso skimage falle
    img_original = np.ones((400, 400), dtype=np.uint8) * 255

mm.show(img_original, figsize=(4, 3))
```

```
PDF de hojas de examen digitalizadas: dados/provas_qrcode_EP.pdf
[INGESTA] Página PDF convertida con éxito: test01.png
[INGESTA] Página PDF convertida con éxito: test02.png
[INGESTA] Página PDF convertida con éxito: test03.png
```

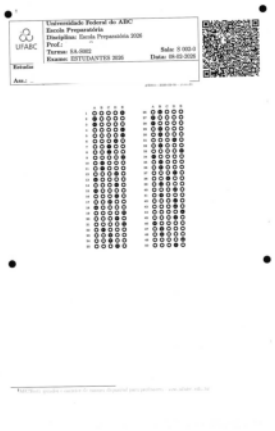


Figura 6.5: *Pipeline* de ingesta de documentos: rasterización adaptativa de páginas PDF a matrices discretas en formato PNG, mostrando la página dos.

6.8.2 Algoritmo de Rectificación de Inclinación (*Deskew*)

La etapa de *deskew* tiene como objetivo estimar y corregir la inclinación global de un documento digitalizado, alineando su contenido con los ejes de la imagen. La Figura 6.7 presenta el flujo completo de procesamiento, desde la imagen original hasta el resultado tras la corrección geométrica. Complementariamente, el simulador de la Figura 6.6 permite visualizar el funcionamiento de la Transformada de Hough y comprender cómo se estima la orientación predominante.

El procedimiento se compone de tres etapas principales:

1. detección de bordes mediante el operador de Canny;
2. estimación de la orientación predominante a través de la Transformada de Hough Lineal;
3. corrección de la inclinación utilizando una transformación afín de rotación.

Tras la rectificación, el documento pasa a presentar una orientación aproximadamente horizontal, favoreciendo las etapas posteriores de segmentación, etiquetado de componentes conexos y reconocimiento de caracteres y marcas.

6.8.3 Modelado Matemático

Las siguientes subsecciones formalizan, en términos matemáticos, las etapas descritas anteriormente, relacionando el gradiente de la imagen, la parametrización de rectas en el espacio de Hough y la matriz de rotación empleada en la corrección geométrica.

6.8.3.1 Detección de Bordes

Inicialmente, la imagen se suaviza mediante un filtro gaussiano, reduciendo el efecto de ruidos de alta frecuencia que pueden generar bordes espurios. Los conceptos de filtrado espacial y convolución se presentaron en el **Capítulo 3**.

A continuación, el operador de Canny estima el gradiente de la imagen. Sea $f(x, y)$ la intensidad de la imagen y α el ángulo de rotación.

La magnitud del gradiente está dada por

$$|\nabla f(x, y)| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}.$$

donde:

- $f(x, y)$ representa la intensidad de la imagen en la posición (x, y);
- $\frac{\partial f}{\partial x}$ y $\frac{\partial f}{\partial y}$ son las derivadas parciales en las direcciones horizontal y vertical;
- $|\nabla f(x, y)|$ es la magnitud del gradiente.

Tras el cálculo del gradiente, el algoritmo aplica la supresión de no máximos (*non-maximum suppression*) y la umbralización por histéresis, produciendo una imagen binaria que contiene los bordes principales del documento.

6.8.3.2 Transformada de Hough

La imagen binaria de bordes se procesa mediante la Transformada de Hough Lineal, cuyo objetivo es detectar estructuras aproximadamente rectilíneas. En lugar de la representación cartesiana de la recta, $y = ax + b$, se utiliza la representación en forma normal,

$$\rho = x \cos\theta + y \sin\theta,$$

donde:

- x e y son las coordenadas de un punto perteneciente a la recta;
- ρ es la distancia perpendicular entre la recta y el origen del sistema de coordenadas de la imagen;
- θ es el ángulo formado entre la normal a la recta y el eje horizontal de la imagen.

En esta representación, cada punto de borde (x, y) genera una curva en el espacio de parámetros (ρ, θ). La intersección de las curvas producidas por puntos pertenecientes a la misma recta origina máximos en una matriz bidimensional denominada **acumulador**. Así, los picos del acumulador corresponden a las rectas predominantes de la imagen, como bordes del documento, líneas de formularios o líneas de texto.

Para estimar la inclinación global del documento, se consideran únicamente las rectas cuyos ángulos satisfacen

$$-45^\circ \leq \theta \leq 45^\circ.$$

Esta restricción elimina orientaciones incompatibles con la disposición esperada del documento y reduce la influencia de rectas verticales o de estructuras irrelevantes. Sea $\theta_1, \theta_2, \ldots, \theta_n$ el conjunto de los ángulos de las rectas seleccionadas. La estimación de la inclinación global se obtiene mediante la mediana,

$$\hat{\theta} = \operatorname{med}(\theta_1, \theta_2, \ldots, \theta_n),$$

donde:

- θ_i es el ángulo de la i-ésima recta detectada por la Transformada de Hough;

- n es el número de rectas consideradas tras el filtrado angular;
- $\hat{\theta}$ es la estimación de la inclinación global del documento.

La mediana se adopta por ser menos sensible a la presencia de detecciones aisladas (*outliers*) que la media aritmética, produciendo una estimación más estable de la orientación predominante.

6.8.3.3 Rotación Afín

Sea $\hat{\theta}$ la inclinación estimada en la etapa anterior. La corrección geométrica consiste en aplicar una transformación afín de rotación alrededor del centro de la imagen, de modo que la orientación predominante pase a coincidir con el eje horizontal. Las transformaciones afines fueron estudiadas en el **Capítulo 2**, junto con las operaciones de traslación, escala, cizallamiento y rotación.

Sea (x, y) la posición de un píxel con respecto al centro de la imagen y (x', y') su posición después de la rotación. La transformación se describe mediante

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = R(\alpha) \begin{bmatrix} x \\ y \end{bmatrix},$$

donde

$$R(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix},$$

siendo:

- (x, y) las coordenadas originales del píxel con respecto al centro de la imagen;
- (x', y') las coordenadas del píxel después de la rotación;
- α el ángulo de rotación aplicado para compensar la inclinación estimada del documento;
- $R(\alpha)$ la matriz de rotación.

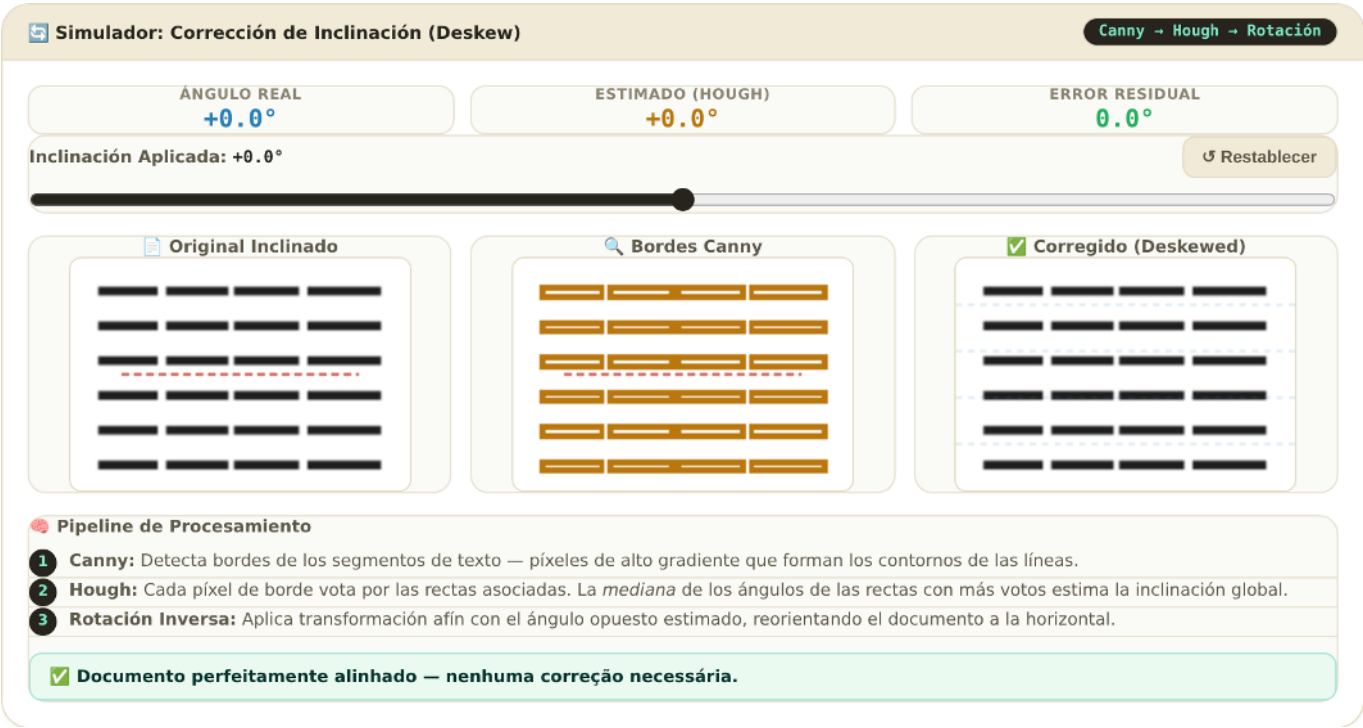
En la práctica, el ángulo aplicado corresponde al opuesto de la inclinación estimada,

$$\alpha = -\hat{\theta},$$

donde $\hat{\theta}$ representa la orientación predominante obtenida mediante la Transformada de Hough.

Como las coordenadas transformadas no siempre coinciden con posiciones enteras de la malla de píxeles, es necesario remuestrear la imagen para determinar los nuevos valores de intensidad. En la implementación presentada en este capítulo, la función `mm.rotate` realiza esta operación utilizando interpolación bicúbica, reduciendo los artefactos de remuestreo y preservando la continuidad visual de bordes y caracteres.

```
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
```



Versión interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap06/sim-06-deskew.html

Figura 6.6: Simulador interactivo de corrección de inclinación (*deskew*): mueva el *slider* para inclinar el documento y observe las tres etapas del *pipeline* — imagen inclinada, bordes Canny y resultado corregido.

```
    return edges, mm.rotate(img, np.median(angulos), interp="bicubic")

# Ejecución del pipeline de deskew
img_edges, img_final = retificar_inclinacao_documento(img_original)

# Exhibición múltiple estandarizada con el formato nativo del libro
mm.show(
    [img_original, img_edges, img_final],
    titles=["Imagen Original", "Bordes de Canny", "Documento Rectificado"],
    cols=3,
    figsize=(12, 4)
)
```

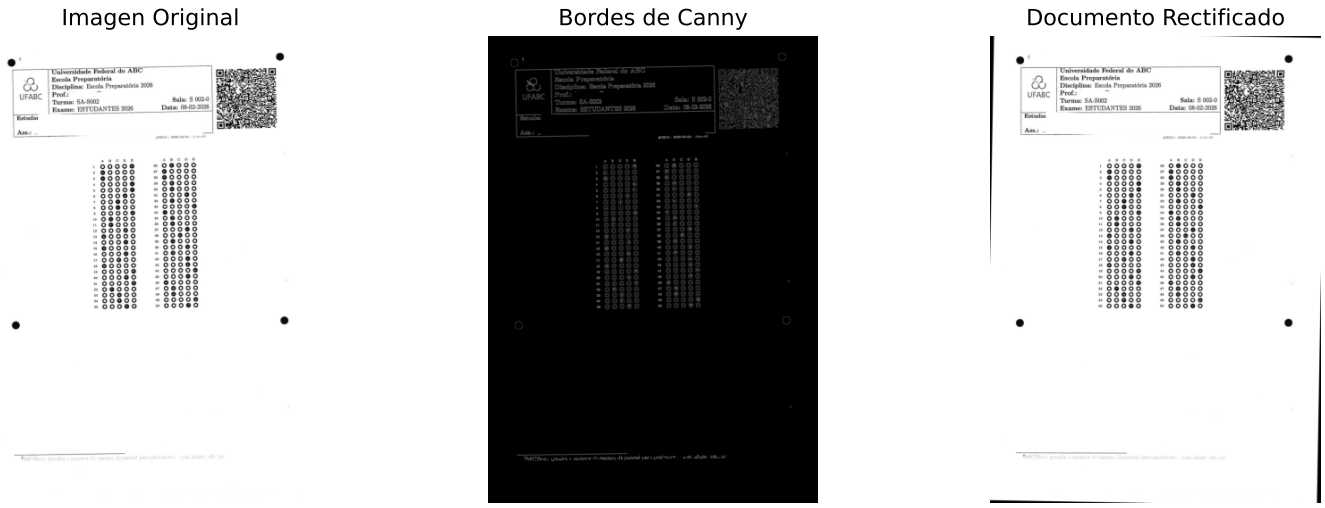


Figura 6.7: *Pipeline* de rectificación axial: exhibición comparativa entre la entrada rotacionada original, el mapa de gradientes estructurales de Canny y el resultado final alineado con fondo normalizado en blanco.

> 🧠 ¿Por qué funciona? — Transformada de Hough
>
> En la Transformada de Hough, cada píxel de borde contribuye con votos para todas las rectas que pueden pasar por su posición. En lugar de seleccionar solo la recta con el mayor número de votos, el algoritmo considera todas las rectas cuya cantidad de votos supera un umbral mínimo y calcula sus respectivos ángulos. La inclinación global del documento se estima entonces mediante la mediana de esos ángulos, una medida robusta ante valores atípicos. Así, las rectas espurias producidas por sombras, ruidos u otros elementos de la imagen ejercen poca influencia sobre la estimación final, siempre que la mayoría de las rectas detectadas corresponda a los bordes del documento.

6.8.4 Limitaciones Prácticas

Aunque presenta un buen rendimiento en condiciones habituales de digitalización, el método depende de la existencia de estructuras lineales suficientemente definidas para ser detectadas mediante la Transformada de Hough, como bordes de página, líneas de formularios o líneas de texto. Su precisión puede verse reducida en imágenes con baja resolución, ruido excesivo, sombras intensas o grandes inclinaciones. En general, los documentos digitalizados con una resolución cercana a 300 DPI y una iluminación homogénea proporcionan resultados adecuados para aplicaciones de OCR y OMR.

La implementación presentada en este capítulo tiene una finalidad didáctica, ilustrando los principios de la corrección automática de inclinación mediante la detección de bordes, la Transformada de Hough y la rotación afín. Al utilizar únicamente la orientación de las estructuras lineales predominantes, el método puede aplicarse a diferentes tipos de documentos, sin depender de marcadores específicos.

En sistemas reales de análisis documental, sin embargo, el alineamiento normalmente utiliza marcadores geométricos previamente conocidos. En el modelo de hoja de respuestas empleado por el ecosistema MCTest, por ejemplo, se utilizan cuatro discos negros de referencia, además de las regiones correspondientes al encabezado, al *QRCode* y a los cuadros de respuestas. La localización de estos elementos permite estimar simultáneamente la rotación, la escala y la traslación de la hoja, haciendo que el registro sea menos sensible a la cantidad de texto, a la ausencia de líneas estructurales y a las variaciones de impresión o digitalización.

Por este motivo, el enfoque basado en la Transformada de Hough se utiliza en este capítulo para introducir los fundamentos del problema, mientras que las etapas posteriores adoptan el alineamiento mediante marcadores geométricos, estrategia predominante en sistemas de OMR y análisis documental.

6.8.5 Detección de Bordes y Contornos

La localización precisa de las regiones de interés es una etapa esencial en sistemas de OMR. En el modelo de hoja de respuestas utilizado en este capítulo, el encabezado y el cuadro de respuestas están contenidos en un rectángulo virtual delimitado por cuatro discos negros posicionados en las esquinas. La identificación de estos marcadores permite localizar la región de interés y corregir distorsiones geométricas introducidas durante la adquisición de la imagen.

El procedimiento se compone de cinco etapas. Inicialmente, se aplica un **cierre morfológico** (dilatación seguida de erosión), operación estudiada en el **Capítulo 4**, utilizando un elemento estructurante en disco (`mm.sedisk(33)`). Esta operación reduce pequeñas discontinuidades y preserva los discos de referencia, haciéndolos más homogéneos. A continuación, la imagen se invierte (`mm.neg`), de modo que los discos pasen a constituir componentes claros sobre fondo oscuro.

En la etapa siguiente, se aplica la operación `mm.edgeoff` (**Capítulo 4**), que elimina componentes conectados a los bordes de la imagen, eliminando artefactos como sombras de digitalización, marcas de corte y otros objetos espurios en los márgenes. Los componentes restantes se analizan entonces a partir de sus contornos y se filtran por propiedades geométricas, como área y circularidad, para identificar los discos candidatos. La implementación del MCTest hace que este proceso sea más robusto al seleccionar, entre todos los candidatos, los cuatro cuyos centros forman un rectángulo con ancho compatible con el de la imagen, reduciendo la ocurrencia de falsos positivos.

Finalmente, los centros de los cuatro discos se ordenan espacialmente (superior izquierdo, superior derecho, inferior izquierdo e inferior derecho) y se utilizan como puntos de control en una transformación de perspectiva (*perspective warp*). Esta transformación rectifica la imagen, produciendo una representación alineada y con dimensiones conocidas, adecuada para las etapas subsiguientes de segmentación y reconocimiento.

Las principales etapas de este *pipeline*, desde el procesamiento morfológico hasta la imagen rectificada, se ilustran a continuación.

```
import cv2
import numpy as np
from morph import mm

# img: imagen en escala de grises de la hoja de prueba
if img_final.ndim == 2:
    img = img_final
else:
    img = mm.gray(img_final)

# 1. Cierre morfológico: preserva los discos oscuros, eliminando todo lo menor que el disco
img_close = mm.close(img, mm.sedisk(41))

# 2. Inversión: los discos oscuros se convierten en componentes claros sobre fondo oscuro
img_neg = mm.neg(img_close)

# 3. Elimina componentes conectados que tocan el borde de la imagen
img_edgeoff = mm.edgeoff(img_neg)

# 4. Extracción de los contornos externos
contornos, _ = cv2.findContours(img_edgeoff, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# 5. Filtrado por área y circularidad, manteniendo solo los 4 discos
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))

# Verificación robusta: interrumpe el pipeline con mensaje claro en lugar de AssertionError
if len(centros) != 4:
    print(f"[AVISO] Se esperaban 4 discos marcadores, encontrado {len(centros)}.")
    print("  Verifique si la imagen es una hoja de respuestas MCTest válida")
    print("  o ajuste los parámetros de circularidad y área mínima.")
    img_retificada = img  # fallback: preserva la imagen sin rectificación
else:
    # 6. Ordenación de los centros: superior-izquierdo, superior-derecho,
    # inferior-izquierdo, inferior-derecho
    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. Rectificación por transformación de perspectiva (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=["Cierre (sedisk 41)", "edgeoff", "Rectificada (warp)"],
        cols=3,
        figsize=(12, 4)
    )

mm.write(img_retificada, "img_beetween_disks.png")
```

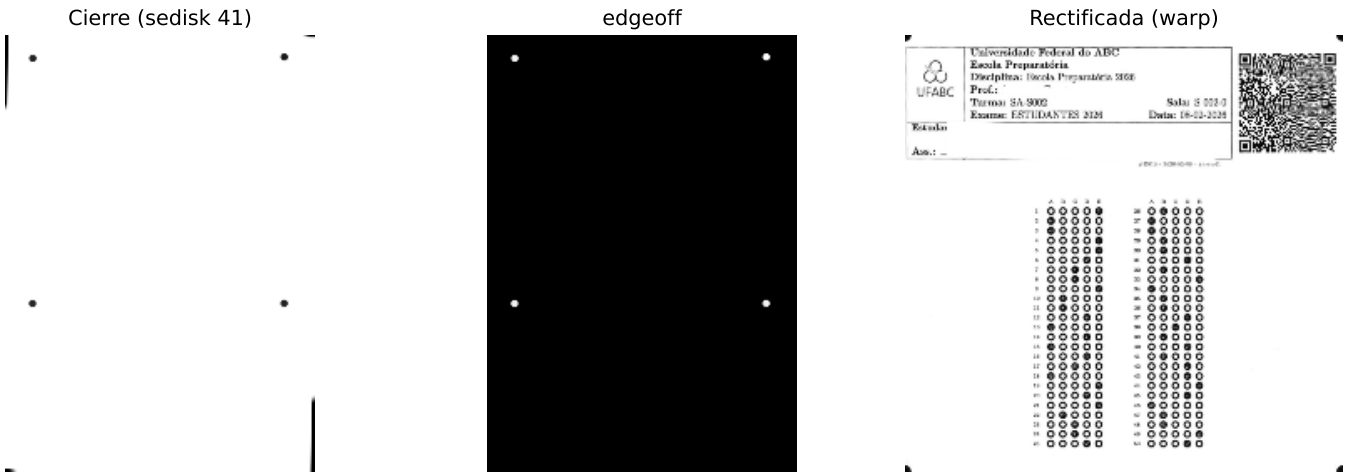
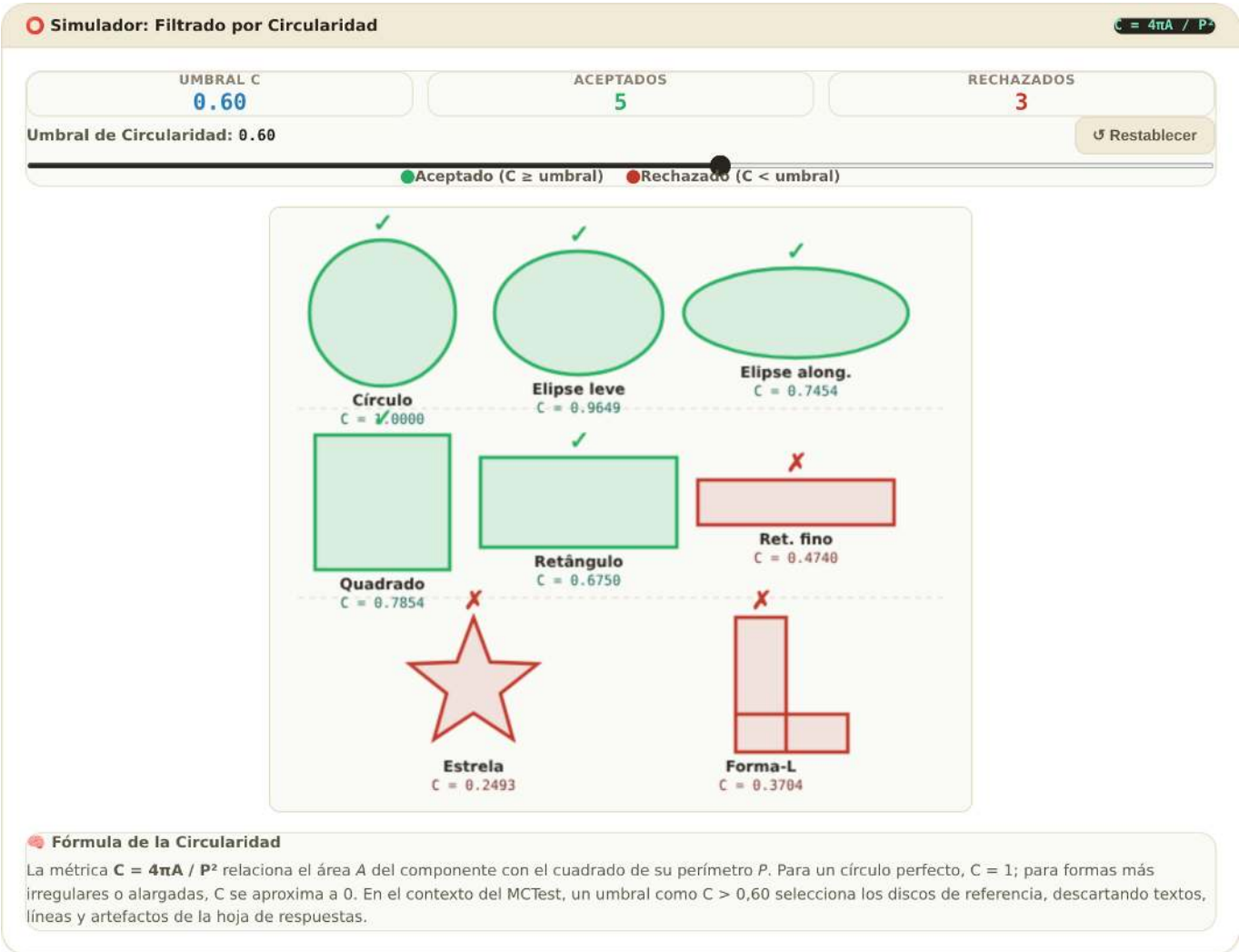


Figura 6.8: Detección de los discos marcadores, extracción de contornos y rectificación por transformación de perspectiva.

¿Por qué funciona? — Del cierre morfológico a la rectificación

Cierre morfológico: la dilatación seguida de erosión rellena pequeñas discontinuidades y suaviza los contornos de los objetos sin alterar significativamente su forma global. Al utilizar un elemento estructurante grande (`sedisk(41)`), los detalles finos, como textos, líneas del formulario y pequeños ruidos, tienden a incorporarse al fondo durante el procesamiento, mientras que los objetos de mayor escala, como los discos de referencia, permanecen preservados y se vuelven más homogéneos.
Circularidad: tras el aislamiento de los componentes candidatos, la métrica $C = \frac{4\pi A}{P^2}$ cuantifica cuánto se aproxima su forma a un círculo. Su valor es igual a 1 para un círculo perfecto y disminuye a medida que el contorno se vuelve más irregular. Así, un umbral como $C > 0{,}8$ permite descartar la mayor parte de los falsos positivos sin recurrir a modelos de aprendizaje. Un simulador de esta métrica se presenta en la Figura 6.9..
Transformación de perspectiva (*perspective warp*): una vez identificados los cuatro discos de referencia, sus centros se utilizan como puntos de control para estimar la transformación proyectiva que mapea la imagen capturada al plano del documento. Esta transformación corrige las distorsiones introducidas por la perspectiva durante la adquisición de la imagen, produciendo una representación frontal con dimensiones conocidas y adecuada para las etapas posteriores de segmentación y reconocimiento.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap06/sim-06-circularidade.html

Figura 6.9: Simulador interactivo de filtrado por circularidad: mueva el *slider* para ajustar el umbral C y observe qué componentes son aceptados (verde) o rechazados (rojo).

6.8.6 Aislamiento, Segmentación y Decodificación del *QRCode*

Tras la rectificación geométrica de la hoja de respuestas, se realiza la detección y la decodificación del *QRCode* presente en el formulario. Este marcador almacena información utilizada por el sistema de OMR (*Optical Mark Recognition*), como la identificación del estudiante, el código de la prueba y su variación, posibilitando la recuperación de la plantilla de respuestas correspondiente en la base de datos. Por seguridad, esta información se cifra antes de la generación del *QRCode*. Así, la secuencia decodificada corresponde a una *cadena* hexadecimal, cuya interpretación se realiza exclusivamente por el sistema MCTest. El procedimiento consta de tres etapas: preprocesamiento morfológico, aislamiento de la región del *QRCode* y decodificación de su contenido.

Inicialmente, la imagen rectificada en escala de grises se binariza mediante la operación `mm.threshold`. A continuación, se aplica una **apertura morfológica** (erosión seguida de dilatación), estudiada en el **Capítulo 4**, utilizando un elemento estructurante cuadrado (`mm.sebox(2)`). Esta operación elimina pequeños ruidos y suaviza imperfecciones sin comprometer la estructura del marcador. Finalmente, la imagen se invierte (`mm.neg`), de modo que el *QRCode* pase a constituir un componente claro sobre fondo oscuro, facilitando la extracción de sus contornos.

La localización del *QRCode* se realiza mediante el análisis de los contornos externos de la imagen binarizada. Entre los componentes detectados, se selecciona aquel con mayor área y geometría aproximadamente cuadrada, descartando los demás elementos impresos de la hoja. A continuación, la región correspondiente se expande con un pequeño margen de seguridad, garantizando la preservación integral del marcador.

El *QRCode* se extrae entonces directamente de la imagen rectificada en escala de grises, preservando su calidad radiométrica. Como esta región generalmente presenta dimensiones reducidas, se aplica un redimensionamiento con interpolación cúbica, aumentando la resolución espacial y favoreciendo la identificación de sus módulos. La lectura se realiza mediante el detector de *QRCode* de OpenCV (`cv2.QRCodeDetector`), que recupera la secuencia de caracteres originalmente codificada.

El flujo completo de este procesamiento, desde el preprocesamiento morfológico hasta la decodificación del *QRCode*, se ilustra en la Figura 6.10. El fragmento mostrado en la salida corresponde solo al inicio de la *cadena* hexadecimal cifrada; su interpretación completa se realiza internamente por MCTest tras la decodificación.

```
import cv2
import numpy as np
from morph import mm

# f: imagen rectificada convertida al tipo correcto de 8 bits (0-255)
f = img_retificada.astype('uint8')

# 1. Umbralización: conversión de la imagen en tonos de gris a binaria
f_thresh = mm.threshold(f)

# 2. Apertura morfológica: elimina pequeños ruidos y suaviza el contorno de los bloques
f_open = mm.open(f_thresh, mm.sebox(2))

# 3. Inversión morfológica: los módulos oscuros se convierten en componentes claros sobre
fondo oscuro
f_inv = mm.neg(f_open)

# 4. Conversión segura a uint8 con escala 0-255
img_uint8 = (f_inv.astype(np.uint8) * 255) if f_inv.max() == 1 else f_inv.astype(np.uint8)

# 5. Detección de contornos externos
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. Filtrado por el mayor contorno con proporción aproximadamente cuadrada
#    (relación de aspecto entre 0.7 y 1.3 descarta rectángulos alargados de la hoja)
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."
    )

# Selecciona el mayor candidato cuadrado por área de bounding box
maior_contorno = max(candidatos, key=lambda c: cv2.boundingRect(c)[2] *
cv2.boundingRect(c)[3])
x, y, w, h = cv2.boundingRect(maior_contorno)

# 7. Expansión de la bounding box con margen de seguridad (evita truncamiento del QRCode)
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)

# 8. Recorte de la región de interés a partir de la imagen original (nítida, en gris)
img_qrcode_final = img_retificada[y1:y2, x1:x2]

# 9. Ampliación para resolución mínima de decodificación (400 px en el lado mayor)
#    cv2.QRCodeDetector requiere módulos con al menos 3-4 px de ancho para decodificar
#    con seguridad; imágenes menores a ~400 px tienden a fallar.
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. Inicialización del detector nativo de QRCode de OpenCV
detector = cv2.QRCodeDetector()

# 11. Detección geométrica y decodificación de los datos textuales
dados, pontos, qrcode_reto = detector.detectAndDecode(img_para_leitura)


# Visualización intermedia: progresión de la binarización al aislamiento del QRCode
mm.show(
    [f_thresh, f_open, f_inv, img_qrcode_final],
    titles=["1. Umbralización", "2. Apertura morfológica", "3. Inversión", "Imagen final"],
    cols=3, figsize=(12, 4)
)

# Validación y salida de los metadatos extraídos
if dados:
    print(f"QRCode decodificado con éxito: \n{dados[:50]}...")
else:
```

```
    raise ValueError(
        "Falha na decodificação do QRCode. "
        "Verifique o limiar, as margens da região ou a qualidade da imagem."
    )
```

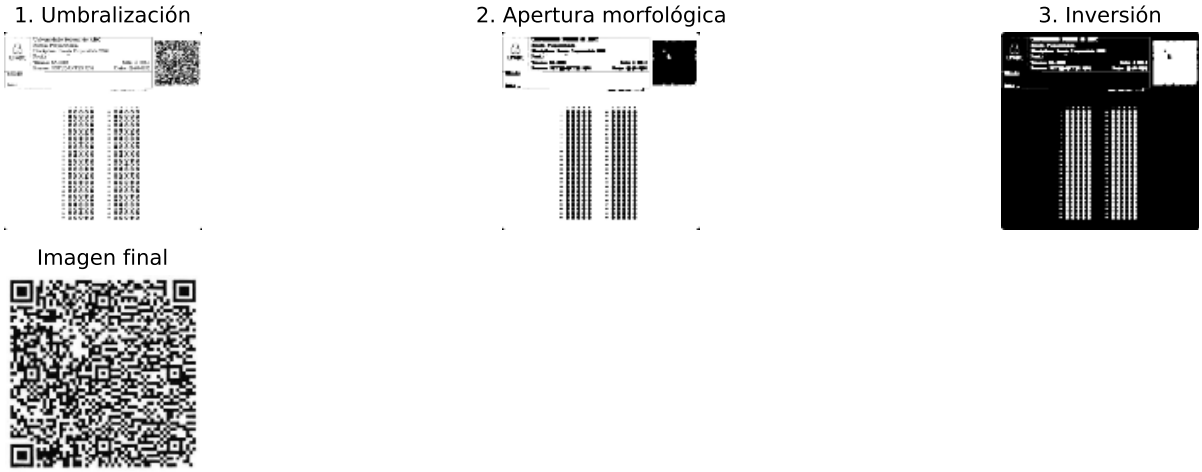


Figura 6.10: Pipeline de procesamiento del *QRCode*: umbralización, apertura morfológica, inversión y recorte final para decodificación.

```
QRCode decodificado con éxito:
325a356b71367266556955646b7233454149624a694f417730...
```

¿Por qué funciona? — Aislamiento y decodificación del *QRCode*

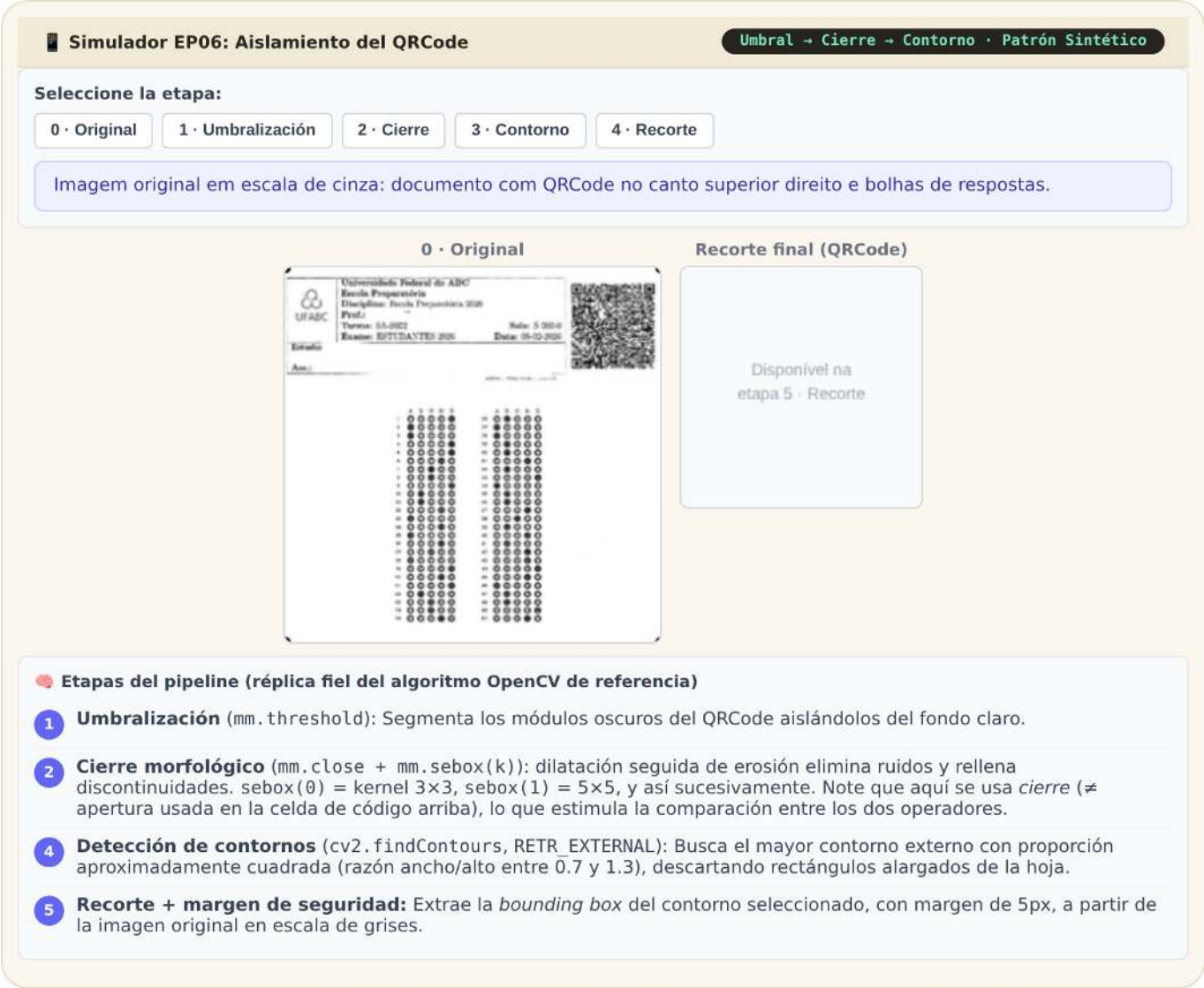
Apertura morfológica: a diferencia del cierre, la apertura (erosión seguida de dilatación) elimina pequeños ruidos y protuberancias sin alterar significativamente la geometría de los objetos más grandes. Así, preserva la estructura del *QRCode* mientras elimina componentes espurios que podrían dificultar su localización.
Selección por geometría: el *QRCode* posee un formato aproximadamente cuadrado ($w/h \approx 1$). La combinación de este criterio con la selección del componente de mayor área descarta líneas del formulario, textos y otros elementos impresos, permitiendo aislar el marcador sin el uso de modelos de aprendizaje.
Redimensionamiento antes de la decodificación: cuando el *QRCode* ocupa pocos píxeles en la imagen, sus módulos se vuelven difíciles de distinguir. El redimensionamiento con interpolación cúbica aumenta la resolución espacial de la región de interés, facilitando la identificación de los patrones del código por el `cv2.QRCodeDetector` y haciendo la decodificación más robusta.

6.8.7 Decodificación de Códigos de Barras

En la sección anterior, la decodificación de *QRCodes* se realizó utilizando el detector nativo de OpenCV (`cv2.QRCodeDetector`), que integra en una única interfaz la detección geométrica del símbolo, su rectificación y la extracción de la información codificada.

Para códigos de barras lineales (1D), como EAN-13, Code 39 y Code 128, una alternativa ampliamente utilizada es la biblioteca `pyzbar`. A diferencia del `QRCodeDetector`, esta soporta diversas simbologías de códigos de barras y también puede emplearse en la lectura de *QRCodes*.

El procedimiento consiste en localizar automáticamente cada símbolo presente en la imagen e interpretar la secuencia de barras y espacios correspondiente, produciendo la cadena de caracteres codificada. Además de los datos decodificados, la biblioteca proporciona información como la simbología identificada y la posición del código en la imagen, permitiendo su posterior validación o procesamiento.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap06/sim-06-qrcode.html

Figura 6.11: Simulador interactivo del *pipeline* de aislamiento del QRCode: navegue por las etapas de filtrado, ajuste el elemento estructurante del cierre morfológico y vea la detección del contorno cuadrado sobre la imagen original de la plantilla.

La Figura 6.12 presenta un ejemplo de código de barras y el resultado de su decodificación utilizando la biblioteca `pyzbar`.

```
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"
)

# Si el archivo no existe localmente, descarga automáticamente desde GitHub
if not os.path.exists(barcode_path):
    print(f"[DESCARGA] Descargando código de barras desde GitHub: {url_github}")
    try:
        os.makedirs(os.path.dirname(barcode_path), exist_ok=True)
        urllib.request.urlretrieve(url_github, barcode_path)
        print("[DESCARGA] ¡Imagen descargada con éxito!")
    except Exception as e:
        print(f"[DESCARGA] Error al descargar el archivo: {e}")

if os.path.exists(barcode_path):
    image = mm.read(barcode_path)
else:
    # Respaldo sintético: genera un patrón de barras verticales que simula un Code-128
    print("[AVISO] Archivo 'datos/barcode.png' no encontrado.")
    print("        Usando imagen sintética para demostración del pipeline.")
    h, w = 100, 400
    img_synth = np.ones((h, w), dtype=np.uint8) * 255
    # Barras oscuras en posiciones regulares (patrón simplificado)
    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)

# Ejecuta el decode de pyzbar
try:
    barcodes = decode(image)
except ImportError:
    print("[ERROR] zbar del sistema no encontrada. Ejecuta: !apt-get install -y libzbar0")
    barcodes = []

if barcodes:
    dados_bc = barcodes[0].data.decode("utf-8")
    tipo = barcodes[0].type
    print(f"Código de barras decodificado con éxito [{tipo}]:\n{dados_bc}")
else:
    print("[INFO] Ningún código de barras detectado en la imagen.")
    print(
        "En imagen sintética esto es esperado - "
        "reemplaza con el archivo real para decodificar."
    )
```

Figura 6.12: Decodificación de código de barras lineal con *pyzbar*: imagen de entrada y datos extraídos.

```
Código de barras decodificado con éxito [EAN13]:
0000000000055
```

6.9 El MCTest como Estudio de Caso: del Prototipo al Sistema en Producción

Hasta este punto, las principales etapas del *pipeline* de procesamiento han sido presentadas y analizadas individualmente, incluyendo la corrección de la inclinación (*deskew*), la detección de marcadores, la rectificación por perspectiva y la lectura de *QRCodes*. Aunque este enfoque facilita la comprensión de cada técnica, las aplicaciones reales exigen la integración de estas etapas en un flujo de procesamiento único y consistente.

El **MCTest** constituye un ejemplo de esta integración. Desarrollado en la UFABC y disponible como software de código abierto, el sistema se utiliza desde 2012 en la corrección automatizada de evaluaciones, ofreciendo soporte para diferentes modelos de hojas de respuesta, plantillas individualizadas y generación automática de informes de desempeño (ZAMPIROLLI, 2023).

A partir de este punto, el enfoque deja de ser la implementación aislada de algoritmos y pasa a ser la organización de estos algoritmos en una aplicación completa. Además de la calidad de los métodos de procesamiento de imágenes, un sistema de esta naturaleza debe cumplir requisitos como robustez frente a diferentes condiciones de adquisición, facilidad de mantenimiento y capacidad de evolución hacia nuevas funcionalidades.

En las próximas secciones, se analizará el módulo de Visión Computacional del MCTest, implementado en el archivo `CVMCTest.py`. El objetivo es mostrar cómo los conceptos presentados a lo largo de este capítulo se combinan en un *pipeline* de procesamiento empleado en una aplicación real.

6.9.1 Obtención y Preparación del Módulo

El archivo `CVMCTest.py` integra el sistema MCTest y, en su versión original, depende de modelos, configuraciones y otros componentes del *framework* Django. Dado que estas dependencias no están disponibles en el entorno utilizado en este capítulo, el módulo debe adaptarse para ejecutarse de forma independiente.

Para ello, el archivo se obtiene directamente del repositorio del proyecto mediante la biblioteca `requests`. A continuación, se emplean comandos `sed` para eliminar las importaciones y dependencias específicas del entorno web, produciendo una versión autocontenida del módulo. Esta adaptación preserva la implementación de los algoritmos de Visión Computacional, permitiendo su ejecución y análisis sin la necesidad de instalar o configurar toda la infraestructura del sistema MCTest.

```python
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)
```

Cada comando `sed` elimina importaciones específicas del entorno Django, haciendo que el archivo `CVMCTest.py` sea utilizable de forma independiente en este capítulo.

```
# elimina lineas 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
```

¿Por qué eliminar las dependencias de Django?

En la implementación original, el archivo `CVMCTest.py` forma parte de una aplicación desarrollada con el *framework* Django y, por ello, importa modelos, configuraciones y otros componentes específicos de ese entorno. Como estos elementos no están disponibles en este capítulo, el módulo no puede importarse directamente.
Los comandos `sed` eliminan únicamente esas dependencias, sin alterar las rutinas de Visión por Computador implementadas en el archivo. De esta manera, el módulo puede ejecutarse de forma independiente, preservando el comportamiento de los algoritmos presentados.
Este procedimiento ilustra un principio importante de la ingeniería de software: separar la lógica de la aplicación de la infraestructura en la que está inserta, facilitando la reutilización, las pruebas y el estudio de componentes específicos.

6.9.2 Extracción del Área de Respuestas

Tras la lectura de la hoja, la función `getAnswerArea` ejecuta automáticamente las etapas de detección de los marcadores de referencia y corrección por perspectiva presentadas en las secciones anteriores. Como resultado, se obtiene una imagen que contiene únicamente la región destinada a las respuestas, alineada y con dimensiones estandarizadas.

Esta estandarización simplifica las etapas posteriores de procesamiento, ya que la ubicación de los campos de marcado pasa a ser conocida e independiente de la posición original de la hoja durante la digitalización.

La Figura 6.13 presenta la hoja de respuestas original en escala de grises, mientras que la Figura 6.14 muestra la región de respuestas obtenida tras la aplicación de la función `getAnswerArea`.

```
import os
from pdf2image import convert_from_path
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"[INGESTA] Página convertida: {myfile0}")
    pages.clear()
    img_color = mm.read(myfile0)
    img_inicial = mm.gray(img_color)
else:
    print("[AVISO] Archivo 'dados/provas_qrcode.pdf' no encontrado.")
    print("        Usando imagen pública skimage.data.page() como sustituto.")
    img_inicial = skdata.page()
```

```
mm.show(img_inicial)
```

```
[INGESTA] Página convertida: extra02.qrcode_p0.png
[INGESTA] Página convertida: extra02.qrcode_p1.png
[INGESTA] Página convertida: extra02.qrcode_p2.png
```

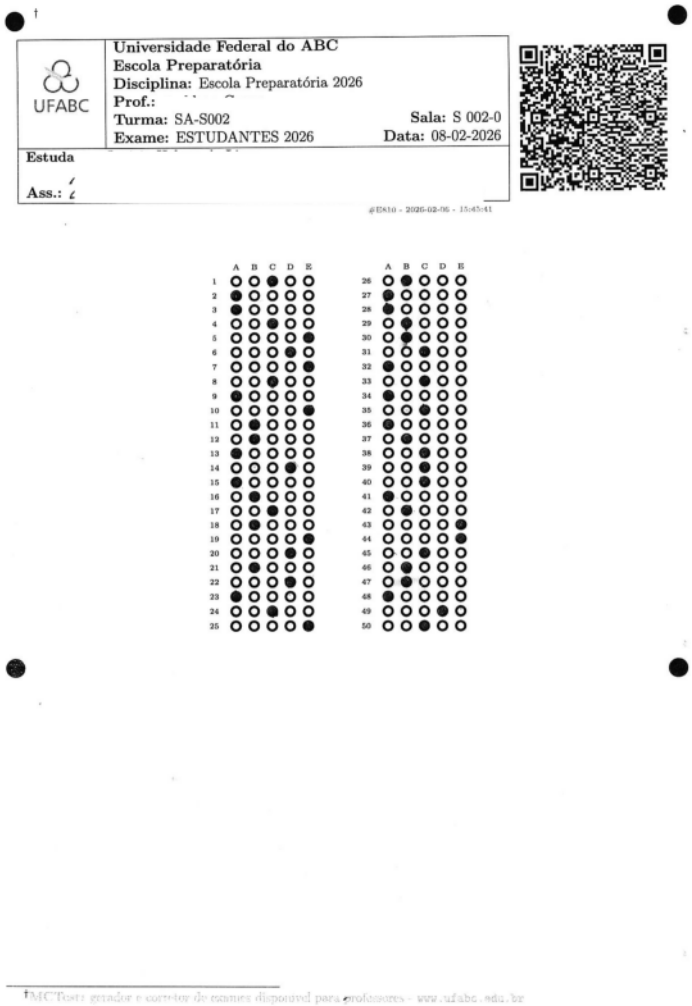


Figura 6.13: Imagen de la hoja de respuestas en escala de grises cargada a partir del PDF rasterizado.

```
import CVMCTest
countPage = 0
img_getAnswerArea = CVMCTest.cvMCTest.getAnswerArea(img_inicial, countPage)
mm.show(img_getAnswerArea)
```

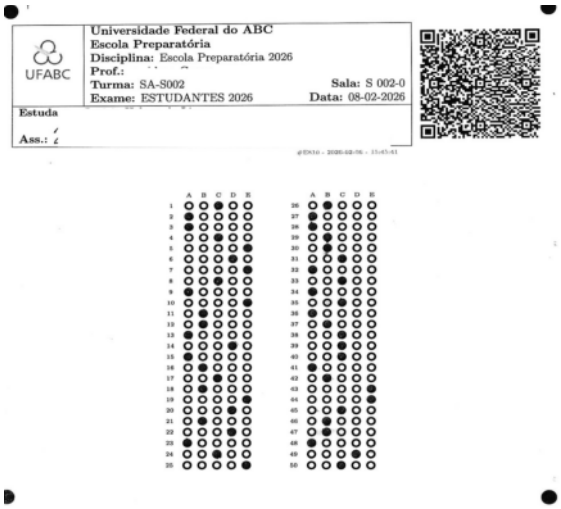


Figura 6.14: Área de respuestas extraída por *getAnswerArea*: región rectificada que contiene los cuadros de marcado.

Nota de compatibilidad: las versiones recientes de NumPy (2.0) eliminaron el alias `np.int0`. Si `CVMCTest.py` utiliza ese tipo, el comando siguiente aplica la corrección directamente en el archivo antes de recargarlo:

```
!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.es/cap06/CVMCTest.py'>
```

6.9.3 Segmentación del *QRCode*

Tras la extracción del área de respuestas, el MCTest realiza dos etapas preparatorias para la lectura de las marcaciones: la segmentación del *QRCode* y la localización de los cuadros que contienen las preguntas.

La función `segmentQRcode` aísla la región de la imagen correspondiente al *QRCode*, presentada en la Figura 6.15.. A continuación, esta región es procesada por la función `getQRCode`, responsable de su decodificación y de la extracción de los metadatos de la prueba.

En caso de que el *QRCode* no pueda ser decodificado, el procesamiento de la hoja continúa normalmente. Las respuestas del estudiante aún se leen y se registran en el archivo CSV de salida; solo la información obtenida a partir del *QRCode*, como la identificación de la prueba o del estudiante, permanece no disponible.

```
import CVMCTest
imgQRcode = CVMCTest.cvMCTest.segmentQRcode(img_getAnswerArea, countPage)
mm.show(imgQRcode)
```

Figura 6.15: Región del *QRCode* aislada por *segmentQRcode* dentro del área de respuestas rectificada.

La función `CVMCTest.cvMCTest.getQRCode(img, countPage)` integra las etapas de segmentación y decodificación del *QRCode*. Internamente, utiliza `CVMCTest.cvMCTest.decodeQRcode(imgQRcode)` para interpretar la *cadena* hexadecimal codificada en el símbolo y construir el diccionario `qr`, además de retornar el indicador lógico `myFlagArea`, que informa si la lectura se realizó con éxito.

El diccionario `qr` reúne los metadatos de la prueba utilizados en las etapas subsiguientes de procesamiento. Sus principales campos son:

- **`date`:** identificador temporal de la prueba, compuesto por la fecha de generación y por un *timestamp* interno del MCTest.
- **`idClassroom`, `idExam` e `idStudent`:** identificadores del aula, de la prueba y del estudiante.
- **`term`:** período lectivo.
- **`stylesheet`:** hoja de estilo utilizada en la generación del formulario.
- **`var1` a `var5`:** cantidad de preguntas en cada nivel de dificultad.
- **`text`:** número de preguntas de desarrollo.
- **`answer`:** número de alternativas por pregunta.
- **`numquest`:** número total de preguntas.
- **`correct`** y **`dbtext`:** campos completados durante la corrección, que contienen la plantilla de respuestas e información adicional.
- **`variations`** y **`variant`:** información sobre las versiones de la prueba.

Estos metadatos identifican la prueba y al estudiante, permitiendo seleccionar la plantilla de respuestas correspondiente y parametrizar las etapas siguientes de lectura y corrección de las respuestas.

```
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'})
```

Estos metadatos permiten identificar la prueba, recuperar la plantilla de respuestas correspondiente y parametrizar las etapas subsiguientes de procesamiento.

Tras la decodificación del *código QR*, el procesamiento regresa al área de respuestas para localizar los recuadros que contienen las marcas del estudiante.

6.9.4 Ubicación de los Cuadros de Respuestas

La imagen producida por `getAnswerArea` contiene toda la región útil de la hoja, incluido el encabezado, donde se encuentra el *QRCode*, y los cuadros destinados a las respuestas. Como la lectura de las marcaciones utiliza únicamente esos cuadros, la región correspondiente al encabezado se descarta mediante el recorte `img_getAnswerArea[300:, :]`.

La Figura 6.16 presenta esta región de interés. Aunque ese recorte sea posteriormente utilizado por la función `findSquares` para localizar los cuadros de respuestas, el MCTest realiza inicialmente la lectura del *QRCode*, pues este contiene los metadatos necesarios para identificar la prueba y configurar las etapas subsiguientes del procesamiento.

```
img_getAnswerArea_aux = img_getAnswerArea[300:,:]
mm.show(img_getAnswerArea_aux)
```

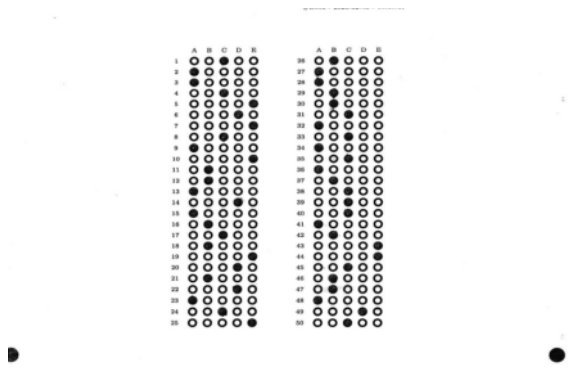


Figura 6.16: Recorte inferior del área de respuestas, concentrando los cuadros de burbujas a ser segmentados.

La función `findSquares` recibe la imagen del área de respuestas y los metadatos almacenados en `qr`, devolviendo las coordenadas de los marcos que delimitan los grupos de preguntas.

Cada elemento de `rectSquares` contiene las coordenadas de los vértices superior izquierdo e inferior derecho de un marco de respuestas, que se utilizarán en la etapa de segmentación de las burbujas.

```
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 Lectura Automática de las Respuestas

Conocidos los metadatos de la prueba (`qr`) y las coordenadas de los marcos de respuestas (`rectSquares`), el MCTest identifica automáticamente las alternativas marcadas por el estudiante.

El siguiente código integra las etapas presentadas anteriormente. Para cada marco delimitado en `rectSquares`, las funciones `setColumns` y `setLines` estiman, respectivamente, el número de alternativas por pregunta y el número de preguntas a partir de la distribución espacial de las burbujas. A continuación, `segmentAnswers` determina la alternativa señalada en cada pregunta y `setAnswersOneLine` reúne los resultados de todos los marcos en el campo `qr['answers']`.

En el modo de operación adoptado en este capítulo, en el que el MCTest se ejecuta de forma independiente de su base de datos, el contenido de `qr['answers']` se compara con la plantilla de corrección almacenada en la primera página del archivo PDF, correspondiente al modelo de prueba sin enunciados utilizado en los ejemplos.

La Figura 6.17 presenta los marcos de respuestas procesados por el algoritmo, mientras que la salida del programa muestra el contenido final de `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)
  # deja las respuestas de cada cuadro en una línea

mm.show(imgQ_all)
print(f"Respuestas leídas de las {len(qr['answers'].split(","))} preguntas: \
      \n{qr['answers'][:-19]}...")
```

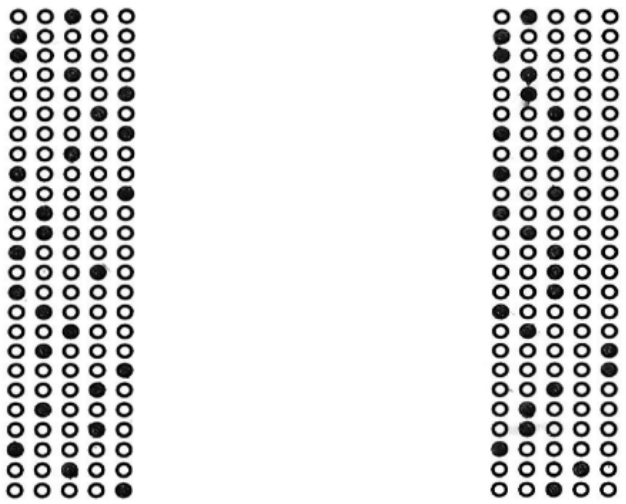


Figura 6.17: Respuestas leídas automáticamente por MCTest tras la segmentación y clasificación de todas las burbujas.

```
Respuestas leídas de las 50 preguntas:
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,...
```

Conectando los Puntos

El campo `qr['answers']` representa el resultado final del *pipeline* de Visión por Computadora presentado en este capítulo. Su obtención integra todas las etapas estudiadas, desde la rasterización del documento y la rectificación geométrica hasta la extracción del área de respuestas, la decodificación del *QRCode*, la localización de los cuadros y la identificación de las alternativas marcadas.
Este *pipeline* ilustra la transición de un prototipo a un sistema en producción. En MCTest, los algoritmos de Visión por Computadora permanecen esencialmente los mismos; las principales diferencias se concentran en aspectos de ingeniería de software, como el manejo de excepciones, el soporte para diferentes modelos de formularios, la integración con la base de datos, la interfaz web y los mecanismos de auditoría y mantenimiento.
En los experimentos de este capítulo, la primera página del archivo PDF contiene la clave de respuestas del examen, mientras que las páginas siguientes corresponden a las hojas de respuestas de los estudiantes. Después de obtener `qr['answers']`, MCTest compara automáticamente las respuestas leídas con la clave para calcular la puntuación de cada estudiante.
En el uso completo del sistema, los resultados de la corrección se consolidan en un archivo CSV y se envían al profesor junto con un archivo comprimido que contiene información auxiliar para auditoría. Entre estos archivos se encuentran los recortes de las preguntas en las que se detectaron múltiples marcaciones u otras situaciones que requieren revisión manual. El profesor puede entonces inspeccionar estas imágenes, decidir la interpretación más adecuada y, si es necesario, actualizar el archivo CSV antes de la importación definitiva de las calificaciones.

6.10 Inspección Industrial Automatizada

La inspección visual automatizada es una aplicación de la Visión por Computadora en la que se analizan imágenes de piezas o productos para verificar el cumplimiento de criterios de calidad previamente definidos. En una línea de producción, las imágenes pueden obtenerse mediante cámaras u otros dispositivos de adquisición y procesarse automáticamente para identificar defectos, medir dimensiones o verificar la presencia de componentes.

La estrategia de inspección depende de las características del producto, del tipo de defecto de interés y de la disponibilidad de una imagen de referencia. En este capítulo se presentan dos enfoques clásicos:

- **Sustracción de imágenes:** compara la imagen de la pieza inspeccionada con una imagen de referencia considerada libre de defectos. Las regiones en las que la diferencia de intensidad supera un umbral se clasifican como posibles defectos. Este enfoque presupone que las imágenes están geométricamente alineadas y han sido adquiridas en condiciones similares de iluminación.
- **Análisis de textura:** utiliza características de la textura de la superficie para identificar regiones cuya apariencia difiere del patrón esperado, sin necesidad de una imagen de referencia. Este enfoque es adecuado para materiales que presentan una textura aproximadamente homogénea, como tejidos, papeles y superficies metálicas.

En las siguientes secciones, estas dos estrategias se ilustran mediante ejemplos construidos a partir de imágenes de la biblioteca `skimage.data`. El objetivo es presentar los principios de funcionamiento de cada enfoque en experimentos que el lector pueda reproducir íntegramente.

6.10.1 Referencias y *Datasets* Públicos

En aplicaciones de inspección industrial, el rendimiento de los algoritmos de detección de defectos se evalúa frecuentemente en *datasets* públicos, que proporcionan imágenes representativas y, en muchos casos, anotaciones de referencia (*ground truth*). En este capítulo, sin embargo, los ejemplos utilizan imágenes sintéticas derivadas de `skimage.data` (ver Figura 6.18), lo que permite reproducir todos los experimentos sin depender de bases de datos externas.

Para estudios más exhaustivos y comparación entre algoritmos, se destacan los siguientes *datasets* públicos:

- **MVTec *Anomaly Detection Dataset* (MVTec AD):** conjunto de imágenes de objetos y texturas, que contiene muestras sin defectos y con defectos, acompañadas de máscaras de segmentación a nivel de píxel para las imágenes anómalas (BERGMANN, 2019). Disponible en: https://www.mvtec.com/company/research/datasets/mvtec-ad.
- **Kolektor *Surface-Defect Dataset* (KolektorSDD):** conjunto de imágenes de componentes industriales con defectos superficiales anotados, utilizado en estudios de detección y segmentación de defectos (TABERNIK, 2020). Disponible en: https://www.vicos.si/resources/kolektorsdd/.
- **NEU *Surface Defect Database*:** conjunto de imágenes de superficies de acero laminado, organizado en seis categorías de defectos superficiales, empleado frecuentemente en la evaluación de métodos de clasificación y detección (SONG, 2013). Disponible en: http://faculty.neu.edu.cn/songkechen/zh_CN/zdylm/263270/list/index.htm.

```
import numpy as np
from skimage import data, color
from morph import mm

# Imagen de referencia (producto sin defecto)
product_color = data.coffee()
product_gray = color.rgb2gray(product_color)

# Inserción de defecto simulado: rayón oscuro de 10×100 px
defect_image = np.copy(product_gray)
defect_image[100:110, 200:300] = 0.1

# Detección por sustracción y umbralización
difference = np.abs(product_gray - defect_image)
defect_threshold = 0.15          # ajustable según la aplicación
detected_defect = (difference > defect_threshold).astype(np.uint8) * 255

mm.show(
    [product_gray, defect_image, detected_defect],
    titles=["Referencia", "Con defecto", "Defecto detectado"],
    cols=3,
```

```
    figsize=(12, 4)
)

status = "Defeito detectado." if detected_defect.any() else "Produto conforme."
print(status)
```

Figura 6.18: Detección de defecto por sustracción de imagen: producto de referencia, imagen con defecto simulado y máscara de anomalía detectada.

```
Defeito detectado.
```

Sustracción de Imágenes: Registro Geométrico y Principio de Funcionamiento

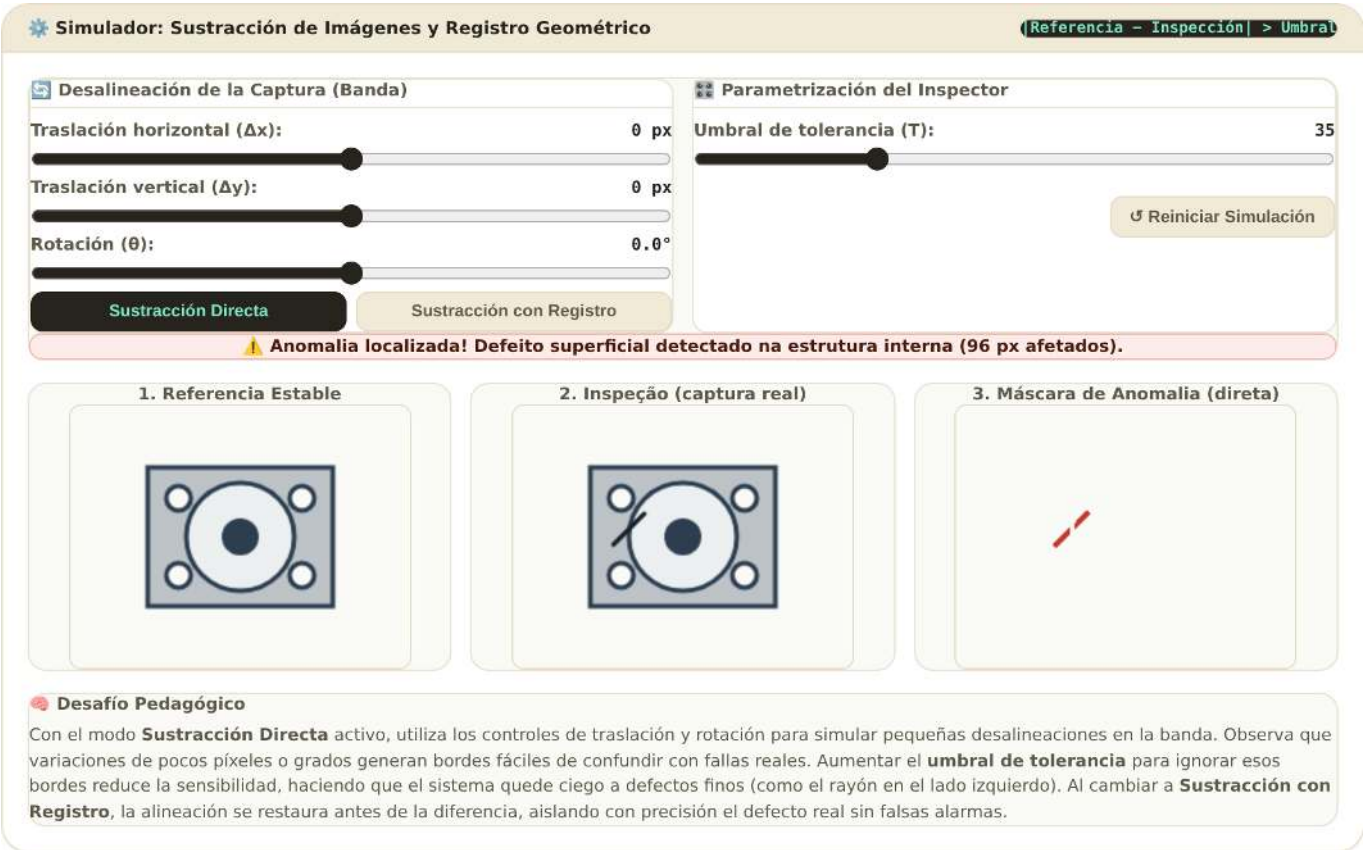
La sustracción de imágenes presupone que la imagen de inspección esté geométricamente alineada con la imagen de referencia. Diferencias de posicionamiento, rotación, escala o perspectiva producen regiones de diferencia que pueden confundirse con defectos.
En aplicaciones con adquisición controlada, este alineamiento se obtiene durante la captura mediante plantillas mecánicas, cintas transportadoras y cámaras fijas, lo que permite comparar directamente imágenes sucesivas. Un ejemplo es la inspección de una caja de herramientas siempre posicionada en la misma orientación para verificar la ausencia de algún elemento.
Cuando este control no es posible, se emplea el **registro de imágenes**, que estima una transformación geométrica para compensar diferencias de traslación, rotación, escala y, cuando sea necesario, perspectiva. Tras el registro, se realiza la comparación píxel a píxel entre las dos imágenes. En regiones sin alteraciones, las diferencias de intensidad tienden a ser cercanas a cero; donde existe un defecto, surgen diferencias locales que pueden destacarse mediante umbralización. La utilización de la diferencia absoluta permite detectar tanto defectos más claros como más oscuros que la referencia.
El rendimiento del método depende principalmente de la calidad del alineamiento geométrico y de la elección del umbral utilizado para separar pequeñas variaciones de adquisición de las diferencias asociadas a los defectos.
La Figura 6.19 ilustra el efecto del desalineamiento entre las imágenes y la importancia del registro geométrico antes de la aplicación de la sustracción.

6.10.2 Detección de Defectos mediante Análisis de Textura

En aplicaciones donde no existe una imagen de referencia, la detección de defectos puede basarse en las características de la textura de la superficie. En este caso, se busca identificar regiones cuya apariencia difiere del patrón predominante del material.

En este ejemplo, se utiliza la **varianza local** como medida de heterogeneidad. Para cada posición de la imagen, se calcula la varianza de los niveles de intensidad en una vecindad de dimensiones fijas. Las regiones con baja varianza tienden a presentar una textura más uniforme, mientras que alteraciones locales, como rayones, manchas o imperfecciones, pueden producir valores más elevados de esta medida.

La Figura 6.20 ilustra este procedimiento utilizando la imagen `skimage.data.brick()`. Inicialmente, se



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap06/sim-06-industrial.html

Figura 6.19: Simulador interactivo de inspección industrial por sustracción de imágenes: controle las distorsiones geométricas de desalineación (registro) y el umbral de detección para observar el impacto en los falsos positivos.

calcula el mapa de varianza local mediante una ventana deslizante. A continuación, se aplica una umbralización para resaltar las regiones cuya varianza excede el valor especificado, identificando posibles áreas de interés para la inspección.

Modelado Matemático

Considérese una imagen en niveles de gris representada por

$$f : \Omega \subset \mathbb{Z}^2 \to \mathbb{R},$$

donde Ω es el dominio de la imagen y $f(x, y)$ representa la intensidad del píxel en las coordenadas (x, y). En imágenes de 8 bits, estas intensidades pertenecen al intervalo $[0, 255]$. En este ejemplo, sin embargo, fueron normalizadas al intervalo $[0, 1]$, sin alterar el funcionamiento del algoritmo.

Para cada posición de la imagen, se considera una vecindad cuadrada $W_{x,y}$ de dimensión 15×15 píxeles.

La media local está dada por

$$\mu(x, y) = \frac{1}{|W_{x,y}|} \sum_{(u,v)\in W_{x,y}} f(u, v),$$

y la varianza local se calcula mediante

$$\sigma^2(x, y) = \frac{1}{|W_{x,y}|} \sum_{(u,v)\in W_{x,y}} f(u, v)^2 - \mu(x, y)^2.$$

En la implementación siguiente, estas dos medias se obtienen mediante la función `cv2.blur`,

```
mean  = cv2.blur(img, (15,15))
mean2 = cv2.blur(img**2, (15,15))
var   = mean2 - mean**2
```

A continuación, se calcula la diferencia entre los mapas de varianza de la imagen de referencia y de la imagen inspeccionada,

$$D(x, y) = \left|\sigma_d^2(x, y) - \sigma_r^2(x, y)\right|,$$

donde $\sigma_r^2(x, y)$ y $\sigma_d^2(x, y)$ son, respectivamente, las varianzas locales de la imagen de referencia y de la imagen que contiene el defecto. Tras la normalización del mapa $D(x, y)$, se aplica una umbralización para obtener la máscara de las posibles anomalías.

```
import numpy as np
import cv2
from skimage import data as skdata
from morph import mm

# Imagen de textura uniforme (ladrillo)
texture = skdata.brick().astype(np.float32) / 255.0

# Inserción de defecto sintético: mancha clara 20×80 px
texture_defect = np.copy(texture)
texture_defect[60:80, 80:160] = 0.95

# Mapa de varianza local (ventana 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)

# Normaliza y umbraliza
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=["Textura original", "Con defecto", "Δ varianza local", "Anomalía detectada"],
    cols=4,
    figsize=(16, 4)
)

status = "Defeito de textura detectado." if mask.any() else "Superfície conforme."
print(status)
```

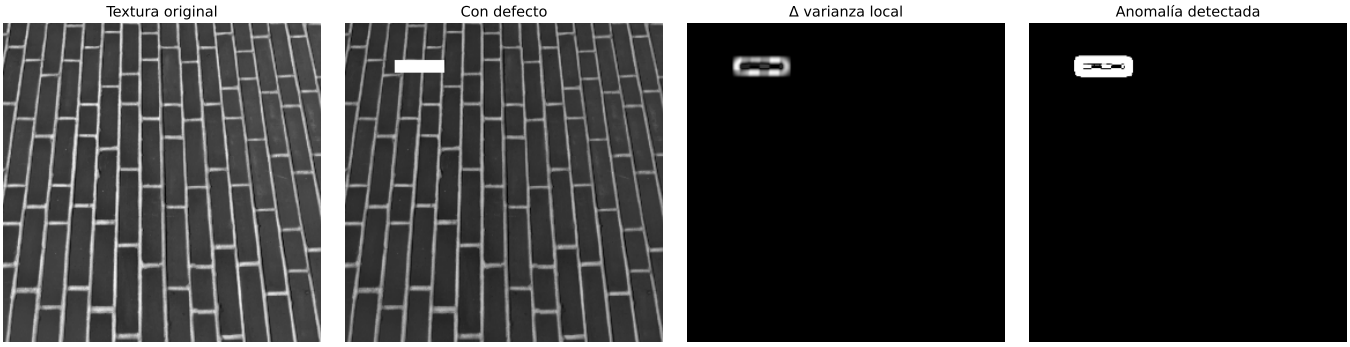


Figura 6.20: Detección de heterogeneidad de textura: mapa de varianza local y máscara de anomalía.

```
Defeito de textura detectado.
```

¿Por qué funciona? — Análisis de Textura

La varianza local mide la dispersión de las intensidades en una vecindad de la imagen. En regiones cuya textura permanece uniforme, esta medida tiende a variar poco. Cuando un defecto modifica el patrón de la superficie, la distribución de las intensidades también se altera, produciendo diferencias en la varianza local.
En este capítulo, la detección se realiza comparando los mapas de varianza de la imagen de referencia y de la imagen con defecto. Tras la normalización, se aplica una umbralización para resaltar las regiones donde esta diferencia supera un valor especificado.
Los principales parámetros del método son el tamaño de la ventana utilizada en el cálculo de la varianza y el umbral empleado en la segmentación. Ventanas más pequeñas son más sensibles a detalles finos, mientras que ventanas más grandes producen mapas más suaves y pueden reducir la respuesta a defectos de pequeñas dimensiones.

6.11 Resumen

En este capítulo se presentaron métodos de Visión por Computador aplicados al análisis de documentos y a la inspección visual automatizada. Las principales técnicas estudiadas fueron:

- **Preprocesamiento de documentos:** aplicación de normalización de fondo, ecualización adaptativa (CLAHE) y umbralización por Otsu para reducir los efectos de iluminación no uniforme y mejorar la segmentación del texto.
- **Reconocimiento óptico de caracteres (OCR):** conversión de imágenes de documentos en texto codificado mediante el Tesseract OCR, evidenciando la influencia del preprocesamiento en la calidad del reconocimiento.
- **Traducción automática:** aplicación de técnicas de procesamiento de lenguaje natural para traducir el texto obtenido por el OCR.
- **Rectificación geométrica de documentos:** utilización del detector de bordes de Canny, de la Transformada de Hough y de transformaciones proyectivas para corregir la perspectiva de documentos digitalizados.
- **Localización y rectificación de formularios:** empleo de operaciones morfológicas, análisis de contornos y transformación de perspectiva para identificar marcadores de referencia y extraer automáticamente regiones de interés.
- **Lectura de códigos bidimensionales y unidimensionales:** detección y decodificación de *QRCodes* y códigos de barras para identificación automática de documentos y metadatos.
- **Reconocimiento óptico de marcas (OMR):** lectura automatizada de formularios y hojas de respuestas, ilustrada por un estudio de caso del sistema MCTest.
- **Inspección industrial:** detección de defectos por comparación con una imagen de referencia y por análisis de la varianza local.

A lo largo del capítulo, los algoritmos se implementaron y evaluaron con imágenes de la biblioteca `skimage.data` y con documentos reales, permitiendo reproducir los experimentos presentados.

Próximos Pasos

Los métodos presentados en este capítulo muestran cómo técnicas de Procesamiento Digital de Imágenes, Visión por Computador y Procesamiento de Lenguaje Natural pueden integrarse en *pipelines* para el análisis automatizado de documentos.

En el próximo capítulo se estudiarán técnicas de **extracción de características** y **reconocimiento de patrones**, con énfasis en descriptores capaces de representar imágenes mediante atributos numéricos para comparación, clasificación y reconocimiento automático. Estos conceptos constituyen la base para los capítulos dedicados al aprendizaje automático y al aprendizaje profundo aplicados a la Visión por Computador.

6.12 🤖 Uso de Gemini Notebook como Tutor Complementario

Como apoyo al estudio de este capítulo, se recomienda la utilización del **Gemini Notebook** como tutor complementario. La herramienta emplea modelos de inteligencia artificial para responder preguntas, elaborar resúmenes y explicar conceptos con base en los documentos proporcionados como fuente de consulta, permitiendo al estudiante revisar el contenido de forma interactiva.

🎓 Estudia con el Tutor Inteligente

🚀 ACCEDER A Gemini Notebook: CAPÍTULO 06

🌐 **Idioma y Lenguaje de Programación**

El proyecto de este capítulo en Gemini Notebook fue construido únicamente con el texto en **portugués** y los ejemplos de código en **Python**. Si estás estudiando desde la edición en inglés o francés, o siguiendo la ruta en C++, las respuestas del tutor pueden no corresponder exactamente a la versión que estás

leyendo.

⚠️ **Uso Crítico de las Respuestas**

Las respuestas generadas por Gemini Notebook se producen automáticamente mediante un modelo de inteligencia artificial y pueden contener omisiones o imprecisiones. Por este motivo, deben utilizarse como material de apoyo, y no como sustituto del estudio del capítulo.
Siempre que surjan dudas, consulta el texto de este libro, ejecuta los ejemplos presentados y, cuando sea necesario, complementa la consulta con libros, artículos científicos y otras fuentes académicas confiables.

6.13 Lista de Ejercicios

Los siguientes ejercicios consolidan los conceptos presentados en este capítulo mediante adaptaciones, experimentos y extensiones de los algoritmos desarrollados a lo largo del texto.

1. **(10%)** Investigue la influencia del ángulo de inclinación en la etapa de *deskew*. Genere versiones rotadas de la imagen `skimage.data.page()` para ángulos entre -10° y 10°, aplique el algoritmo presentado en el capítulo y compare el ángulo estimado con el ángulo utilizado en la rotación. Presente los resultados en una tabla y discuta la precisión del método.

2. **(15%)** Aplique normalización de fondo y CLAHE (utilizando al menos tres combinaciones de `clipLimit` y `tileGridSize`) a la imagen `skimage.data.page()` degradada artificialmente con gradiente de iluminación y sombra lateral. Segmente cada versión mediante el método de Otsu y compare los resultados utilizando el número de componentes conexos espurios y la métrica IoU en relación con una máscara de referencia construida manualmente.

3. **(15%)** Investigue la sensibilidad del filtrado por circularidad, $C = \frac{4\pi A}{P^2}$, en la detección de los marcadores circulares. Utilizando el simulador de la Figura 6.9, genere discos sintéticos con ruido geométrico creciente y evalúe los umbrales $C \in \{0{,}5,\ 0{,}6,\ 0{,}7,\ 0{,}8\}$. Presente una tabla que relacione el umbral con el número de falsos positivos y falsos negativos y discuta el compromiso entre sensibilidad y especificidad.

4. **(15%)** A partir de los cuatro marcadores detectados, implemente la rectificación por perspectiva utilizando `cv2.getPerspectiveTransform` y `cv2.warpPerspective`. A continuación, perturbe artificialmente las coordenadas de los puntos de control con ruido gaussiano de desviación estándar $\sigma \in \{1, 3, 5\}$ píxeles y evalúe el error de reproyección obtenido tras la homografía inversa.

5. **(15%)** Adapte el *pipeline* de adquisición y rectificación desarrollado en este capítulo para procesar documentos que contengan códigos de barras lineales en sustitución de los *QRCodes*. Rasterice el PDF con `pdf2image` a 300 DPI, aplique el *deskew*, decodifique el símbolo con `pyzbar` y presente la imagen rectificada junto con la secuencia de caracteres obtenida.

6. **(15%)** Extienda la lectura de burbujas del MCTest para identificar tres situaciones: **OK**, **BLANCO** (ninguna alternativa marcada) y **DOBLE MARCA** (dos o más alternativas por encima de un umbral de relleno). Evalúe al menos tres valores de dicho umbral, presente los resultados en un `pandas.DataFrame` y discuta su influencia en la clasificación de las respuestas.

7. **(15%)** Construya un *pipeline* de inspección industrial que combine sustracción de imágenes y análisis de varianza local de la textura sobre un conjunto de imágenes sintéticas que contengan defectos simulados. Para cada imagen, genere una máscara de referencia (*ground truth*), calcule la métrica IoU (*Intersection over Union*) de los dos enfoques para diferentes umbrales de decisión y presente los resultados en tablas y visualizaciones producidas con `mm.show`.

8. **(Bônus – 10%)** Implemente manualmente la estimación del ángulo de inclinación sin utilizar `cv2.HoughLines` ni `cv2.HoughLinesP`. A partir del mapa de bordes obtenido por el detector de Canny, construya el acumulador de la Transformada de Hough, $\rho = x\cos\theta + y\sin\theta$, para $\theta \in [-45^\circ, 45^\circ]$, identifique los máximos del acumulador y estime la inclinación mediante la mediana de las rectas

detectadas. Compare los resultados con la implementación de OpenCV y discuta la influencia de rectas espurias en la estimación final.

Referencias del Capítulo

La fundamentación teórica y los estudios de caso presentados en este capítulo se apoyan en las siguientes referencias:

- Gonzalez (2018), para los fundamentos de detección de bordes, umbralización, segmentación, operaciones morfológicas, reconocimiento óptico de caracteres y transformaciones geométricas aplicadas al análisis de documentos.
- Szeliski (2022), para la Transformada de Hough, el registro y alineamiento de imágenes, las transformaciones proyectivas (homografías) y los fundamentos de la inspección visual automatizada.
- Bradski (2008), para la utilización de la biblioteca OpenCV en las etapas de detección de bordes, Transformada de Hough, transformaciones geométricas, análisis de contornos y decodificación de *códigos QR*.
- Smith (2007) y Smith (2013), para la arquitectura, el funcionamiento y la evolución del mecanismo de reconocimiento óptico de caracteres **Tesseract OCR**, empleado en los ejemplos de OCR presentados en este capítulo.
- Bahdanau (2015) y Vaswani (2017), para los fundamentos de la traducción automática basada en redes neuronales, incluidos los mecanismos de atención y las arquitecturas *transformer*.
- Zampirolli (2023), para la descripción del sistema MCTest, utilizado como estudio de caso de un *pipeline* completo para la lectura y corrección automatizada de hojas de respuestas.
- Bergmann (2019), Tabernik (2020) y Song (2013), para los *conjuntos de datos* públicos de inspección industrial **MVTec AD**, **KolektorSDD** y **NEU Surface Defect Database**, utilizados como referencia para la evaluación y comparación de algoritmos de detección de defectos.

Executar Colab Abrir GitHub

6.14 💻 Parte Práctica con Ejercicios de Programación

Los ejercicios de programación (EP) de esta sección complementan los conceptos presentados a lo largo del Capítulo 6 mediante la implementación de algoritmos relacionados con la inspección industrial y el análisis de documentos. El objetivo es consolidar los fundamentos estudiados, reproduciendo, a escala reducida, etapas de un *pipeline* típico de Visión por Computadora.

A diferencia de los capítulos anteriores, cuyos ejercicios enfatizaban operaciones más directamente relacionadas con los datos de imagen, los EP de este capítulo se centran en las **magnitudes intermedias** producidas durante el procesamiento, como áreas, perímetros, circularidad, ángulos de rectas, grados de relleno de burbujas, mapas de varianza y mapas de diferencia. Este enfoque permite comprender y validar cada etapa del *pipeline* de forma independiente, sin depender de bibliotecas especializadas para la adquisición de imágenes, detección de marcadores o decodificación de códigos — con excepción del ejercicio de cierre del capítulo (EP06_08), que intencionalmente introduce el uso de OpenCV para la segmentación y la decodificación real de un *QRCode*, cerrando el ciclo entre los conceptos teóricos y las herramientas empleadas en la práctica.

Los ejercicios siguen la misma secuencia conceptual del capítulo, en orden creciente de complejidad. Inicialmente, se abordan métricas de evaluación de segmentación, utilizadas para cuantificar la calidad de máscaras binarias. A continuación, se estudian criterios geométricos para la selección de marcadores, clasificación de marcas en formularios y estimación de la inclinación de documentos mediante la Transformada de Hough. En la parte final, los ejercicios exploran la normalización de iluminación, la detección de defectos

por análisis de textura y la integración entre registro geométrico y sustracción de imágenes en un *pipeline* simplificado de inspección industrial.

Cada ejercicio representa una etapa aislada de un sistema real de Visión por Computadora, permitiendo validar individualmente conceptos que, en aplicaciones industriales, se combinan en un único *pipeline* de inspección.

Leyenda de Dificultad

Nivel	Significado	EP
🟢	Muy fácil / fácil — implementación de un único concepto o algoritmo simple	EP06_01, EP06_02
🟡	Fácil–medio — tratamiento de múltiples casos o utilización de criterios estadísticos simples	EP06_03, EP06_04
🟠	Medio — procesamiento matricial punto a punto	EP06_05
🔴	Difícil — procesamiento matricial con operaciones en vecindad (ventana deslizante)	EP06_06
🟣	Muy difícil — integración de múltiples etapas de un *pipeline* de Visión por Computadora	EP06_07
⚫	Especial — uso de biblioteca especializada (`cv2`) para segmentación geométrica y decodificación real de código de barras/QRCode	EP06_08

❗ Directrices para la Resolución de los Ejercicios de Programación

Salvo indicación en contrario, todos los ejercicios utilizan la convención de coordenadas matriciales `[fila][columna]`, con origen en $(0, 0)$ en la esquina superior izquierda de la imagen.
Cuando sea necesario realizar un redondeo numérico, se debe utilizar el redondeo estándar al entero más cercano (*round half away from zero*, con `np.floor(img + 0.5)`). Las comparaciones con umbrales (por ejemplo, circularidad, varianza, diferencia de intensidad o grado de relleno) deben considerarse **estrictas** (`>`), excepto cuando el enunciado especifique explícitamente otro criterio.
Cada ejercicio ha sido elaborado para enfatizar un concepto específico presentado en el capítulo. Se recomienda implementar inicialmente la solución de forma directa y, solo después de su validación, buscar alternativas más eficientes o más generales.

🎯 Objetivo de este Cuaderno

Este cuaderno ha sido elaborado para apoyar el desarrollo, la validación y las pruebas de las soluciones de los **Ejercicios de Programación (EPs)** en un entorno interactivo, como Google Colab o Jupyter Notebook. Tras verificar el funcionamiento de la implementación con los casos de prueba presentados, el código puede ser enviado a Moodle para la evaluación oficial.

Download

Ejecute la celda siguiente para obtener los archivos `morph.py` y `testsuite.py`, utilizados por los ejercicios de este capítulo.

```
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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Ejecutando las pruebas

Tras implementar la solución, ejecute `TestSuite("EP06_01.extensión").run()` en una nueva celda, reemplazando `extensión` por el lenguaje utilizado (`.py`, `.java`, `.c`, `.cpp`, `.js` o `.r`). El sistema obtiene automáticamente los casos de prueba del repositorio del curso, ejecuta el programa y presenta el resultado de la evaluación.

En Python, también es posible probar la solución directamente a partir de una *cadena*, sin necesidad de guardar el código en un archivo. Para ello, almacene el programa en una variable y utilice el método `run_code`:

```
codigo = """
# ... su código aquí ...
"""

TestSuite("EP06_01").run_code(codigo)
```

6.14.1 EP06_01 🟢 Evaluación de Segmentación por IoU (*Intersection over Union*)

A lo largo de este capítulo, diversas etapas del *pipeline* producen **máscaras binarias**, como en la segmentación de documentos, la localización de *QRCodes* y la detección de defectos. Para evaluar objetivamente la calidad de estas segmentaciones, es necesario compararlas con una máscara de referencia (*ground truth*).

Una de las métricas más utilizadas para este fin es la **IoU** (*Intersection over Union*, o Intersección sobre Unión), definida como la razón entre el área de intersección y el área de unión de dos máscaras binarias. Cuanto mayor sea el valor de la IoU, mayor será la concordancia entre la segmentación producida por el algoritmo y la referencia.

6.14.1.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (número de filas) y C (número de columnas).
2. **Máscara de referencia:** Leer los $L \times C$ elementos binarios (0 o 1) de la matriz `ref`.
3. **Máscara predicha:** Leer los $L \times C$ elementos binarios (0 o 1) de la matriz `pred`.
4. **Intersección:** Contar el número de posiciones (i, j) para las cuales `ref[i][j] = 1` y `pred[i][j] = 1`.
5. **Unión:** Contar el número de posiciones (i, j) para las cuales `ref[i][j] = 1` o `pred[i][j] = 1`.
6. **Caso degenerado:** Si la unión es igual a 0, definir IoU = 1,0, ya que ambas máscaras están vacías.
7. **Cálculo:** Si la unión es mayor que cero, calcular

$$\text{IoU} = \frac{|\text{Interseccion}|}{|\text{Union}|}.$$

8. **Clasificación:** Determinar la clasificación cualitativa utilizando el valor de IoU **antes** del redondeo.
9. **Redondeo:** Mostrar la IoU con cuatro decimales.
10. **Salida:** Imprimir, en este orden, la intersección, la unión, la IoU y la clasificación.

6.14.1.2 📌 Restricciones Computacionales

- Si la unión es igual a 0, no se debe realizar la división; la IoU debe definirse como 1,0.
- Los rangos de clasificación utilizan comparaciones no estrictas ($\geq$).
- La clasificación debe realizarse utilizando el valor de la IoU en precisión completa, antes del redondeo para la visualización.

6.14.1.3 🧠 Fundamentación Teórica

La IoU se define por

$$\text{IoU} = \frac{|R \cap P|}{|R \cup P|},$$

donde:

- R representa el conjunto de píxeles que pertenecen a la máscara de referencia;
- P representa el conjunto de píxeles que pertenecen a la máscara predicha;
- $|R \cap P|$ corresponde al número de píxeles que pertenecen simultáneamente a ambas máscaras;
- $|R \cup P|$ corresponde al número de píxeles que pertenecen al menos a una de las máscaras.

Rango de IoU	Clasificación	Interpretación
IoU $\geq$ 0,90	`EXCELENTE`	Concordancia muy elevada entre las máscaras.
0,70 $\leq$ IoU $<$ 0,90	`BUENO`	Pequeñas diferencias entre las máscaras.
0,50 $\leq$ IoU $<$ 0,70	`ACEPTABLE`	Concordancia parcial entre las máscaras.
IoU $<$ 0,50	`MALO`	Baja concordancia entre las máscaras.

La IoU depende únicamente de la superposición entre las máscaras y, por lo tanto, es independiente del tamaño de la imagen.

6.14.1.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: entero L.
- Línea 2: entero C.
- Siguientes L líneas: elementos binarios (0 o 1) de la matriz `ref`.
- Siguientes L líneas: elementos binarios (0 o 1) de la matriz `pred`.

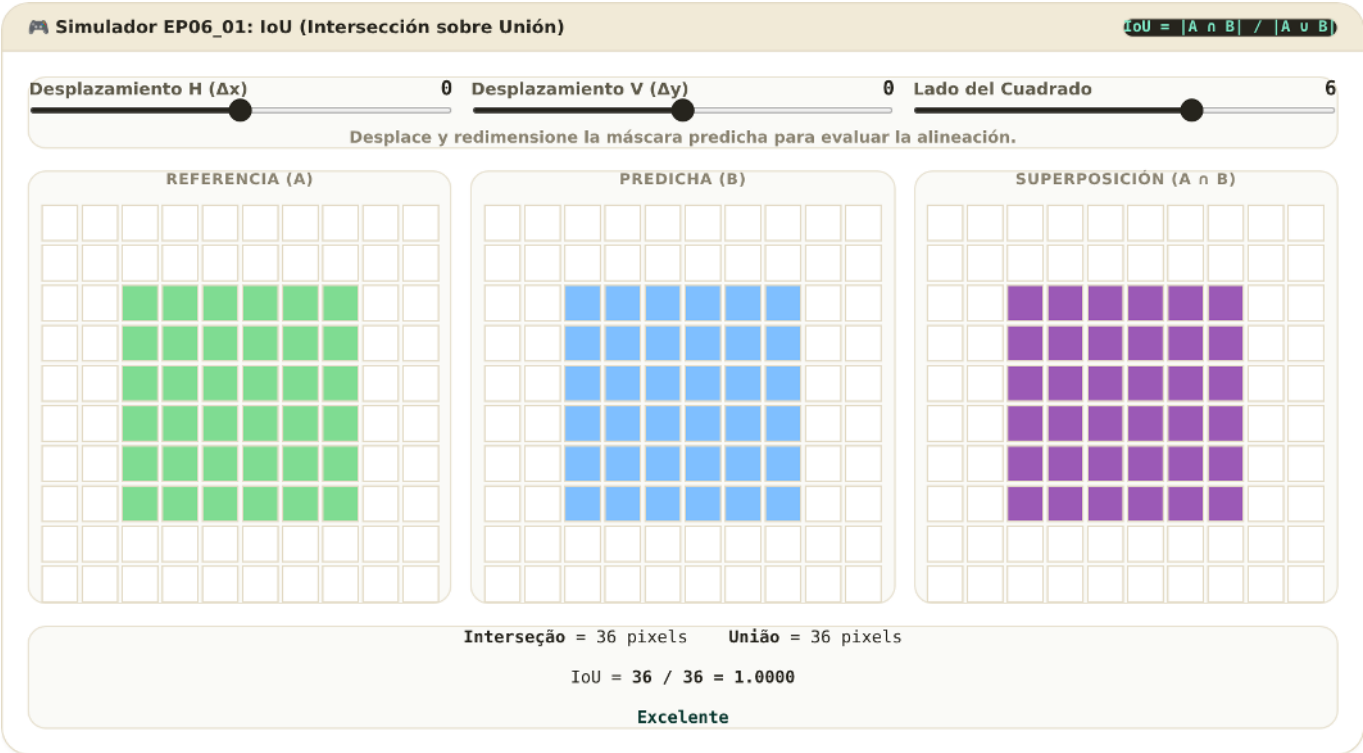
Salida:

- Línea 1: `Interseccion: X`
- Línea 2: `Union: Y`
- Línea 3: `IoU: Z`
- Línea 4: `Clasificacion: NOMBRE`

El valor de `IoU` debe imprimirse con cuatro decimales.

6.14.1.5 📌 Ejemplos

Entrada	Salida	Observación
2 2 1 1 0 0 1 0 0 0	Interseccion: 1 Union: 2 IoU: 0.5000 Clasificacion: ACEPTABLE	La mitad de la región de referencia fue segmentada correctamente.
2 2 0 0 0 0 0 0 0 0	Interseccion: 0 Union: 0 IoU: 1.0000 Clasificacion: EXCELENTE	Ambas máscaras están vacías; por convención, IoU = 1,0.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap06/sim-ep0601.html

Figura 6.21: Simulador EP06_01: IoU entre máscara de referência e máscara predita

```
%%writefile EP06_01.py
# Código Python
```

```
Overwriting EP06_01.py
```

```
TestSuite("EP06_01.py").run()
```

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

6.14.2 EP06_02 🟢 Filtro de Marcadores por Circularidad

Tras la segmentación de una imagen, es común que se identifiquen diversos componentes conexos. En aplicaciones como la rectificación de documentos, solo algunos de estos componentes corresponden a los marcadores de referencia utilizados para el alineamiento de la imagen. Un criterio empleado con frecuencia para seleccionar estos marcadores es la **circularidad**, que mide cuán cercana es la forma de un componente a un círculo.

En este ejercicio, cada componente se describe por su área A y su perímetro P. El objetivo es calcular su circularidad y decidir, a partir de un umbral proporcionado, si el componente debe ser aceptado o rechazado como candidato a marcador.

6.14.2.1 📋 Directrices de Implementación

1. **Cantidad:** Leer el entero N (número de candidatos) y el umbral de circularidad C_{umbral} (número real).
2. **Datos de los candidatos:** Para cada uno de los N candidatos, leer el área A (entero) y el perímetro P (número real).
3. **Circularidad:** Calcular $C = \frac{4\pi A}{P^2}$, donde:

- A es el área del componente;
- P es el perímetro del componente;
- C es la circularidad.

4. **Caso degenerado:** Si $P = 0$, considerar $C = 0$ y clasificar directamente al candidato como `RECHAZADO`.
5. **Clasificación:** Si $C > C_{\text{umbral}}$, clasificar al candidato como `ACEPTADO`; en caso contrario, clasificarlo como `RECHAZADO`.
6. **Redondeo:** Mostrar el valor de C con cuatro decimales.
7. **Salida:** Para cada candidato, imprimir el valor de C seguido de la clasificación. Al final, imprimir el número total de candidatos aceptados.

6.14.2.2 📌 Restricciones Computacionales

- Utilizar la constante π de la biblioteca estándar del lenguaje (por ejemplo, `math.pi`), sin aproximaciones.
- La comparación debe realizarse con el valor de C en precisión completa, antes del redondeo para la visualización.
- El criterio de aceptación es estricto ($C > C_{\text{umbral}}$).
- Si $P = 0$, no se debe realizar la división.

6.14.2.3 🧠 Fundamentación Teórica

La circularidad es un descriptor geométrico definido por $C = \frac{4\pi A}{P^2}$, donde:

- A es el área del componente;
- P es el perímetro del componente;
- C es la circularidad.

Para un círculo perfecto, $C = 1$. A medida que la forma se vuelve más alargada o irregular, el perímetro crece más rápidamente que el área, reduciendo el valor de C.

Forma	Circularidad aproximada	Interpretación
Círculo	1,0000	Forma circular.
Cuadrado	0,7854	Forma aproximadamente compacta.
Forma alargada o irregular	$C \ll 1$	Baja circularidad.
$P = 0$	0 (convención adoptada)	Contorno degenerado.

La circularidad es invariante a la traslación, la rotación y la escala, y se utiliza ampliamente para distinguir componentes aproximadamente circulares de otros formatos.

6.14.2.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: entero N.
- Línea 2: número real C_{umbral}.
- Siguientes N líneas: área A (entero) y perímetro P (real), separados por espacio.

Salida:

- Una línea para cada candidato, en el formato `C ACEPTADO` o `C RECHAZADO`, con C presentado con cuatro decimales.
- Última línea: `Total aceptados: X`.

6.14.2.5 📌 Ejemplos

Entrada	Salida	Observación
3 0.6 78 31.4 100 40 50 60	0.9941 ACEPTADO 0.7854 ACEPTADO 0.1745 RECHAZADO Total aceptados: 2	Candidato aproximadamente circular, forma compacta y forma alargada.
1 0.9 10 0	0.0000 RECHAZADO Total aceptados: 0	Perímetro nulo: contorno degenerado.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap06/sim-ep0602.html

Figura 6.22: Simulador EP06_02: Filtro de Marcadores por Circularidad

```
%%writefile EP06_02.py
# Código Python
```

```
Overwriting EP06_02.py
```

```
TestSuite("EP06_02.py").run()
```

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

6.14.3 EP06_03 🟡 Clasificación de Marcaciones en Hojas de Respuesta (OMR)

Tras la corrección de la hoja y la segmentación de los cuadros de respuestas, el MCTest estima, para cada burbuja, un **grado de relleno**, representado por un valor entre 0 y 100. A partir de estos valores, el sistema

debe determinar automáticamente la alternativa marcada, identificando también preguntas en blanco y casos de múltiples marcaciones.

En este ejercicio, implementará esta etapa de decisión del *pipeline* de OMR. La clasificación depende de un umbral de relleno: pequeñas variaciones en este valor pueden alterar el resultado de la lectura automática.

6.14.3.1 📋 Directrices de Implementación

1. **Parámetros:** Leer los enteros Q (número de preguntas) y K (número de alternativas por pregunta, con $2 \leq K \leq 26$) y el umbral de relleno Th (número real entre 0 y 100).
2. **Grados de relleno:** Para cada una de las Q preguntas, leer los K valores reales correspondientes a las alternativas `A`, `B`, `C`, …, en el orden de entrada.
3. **Conteo de marcaciones:** Para cada pregunta, contar cuántas alternativas poseen un grado de relleno **estrictamente mayor** que Th.
4. **Clasificación:**
   - Si ninguna alternativa excede Th, clasificar la pregunta como `BRANCO`.
   - Si exactamente una alternativa excede Th, imprimir la letra correspondiente (`A`, `B`, `C`, …).
   - Si dos o más alternativas exceden Th, clasificar la pregunta como `DUPLA_MARCACAO`.
5. **Salida por pregunta:** Imprimir, en el orden de lectura, la clasificación de cada pregunta.
6. **Totales:** Al final, imprimir el número de preguntas `OK` (una única marcación), `BRANCO` y `DUPLA_MARCACAO`.

6.14.3.2 📌 Restricciones Computacionales

- **Comparación estricta:** solo los valores mayores que Th se consideran marcaciones válidas; los valores exactamente iguales al umbral no deben contabilizarse.
- **Letras de las alternativas:** el índice 0 corresponde a la alternativa `A`, el índice 1 a la alternativa `B` y así sucesivamente.
- **Múltiples marcaciones:** siempre que dos o más alternativas excedan el umbral, la clasificación debe ser `DUPLA_MARCACAO`, independientemente de los respectivos grados de relleno.

6.14.3.3 🧠 Fundamentación Teórica

Situación	Clasificación	Interpretación
Exactamente una alternativa por encima del umbral	Letra de la alternativa	Respuesta válida
Ninguna alternativa por encima del umbral	`BRANCO`	Pregunta no respondida
Dos o más alternativas por encima del umbral	`DUPLA_MARCACAO`	Respuesta ambigua

El umbral de relleno controla la sensibilidad del algoritmo. Valores muy bajos tienden a aumentar el número de `DUPLA_MARCACAO`, mientras que valores muy altos pueden aumentar la cantidad de preguntas clasificadas como `BRANCO`.

6.14.3.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

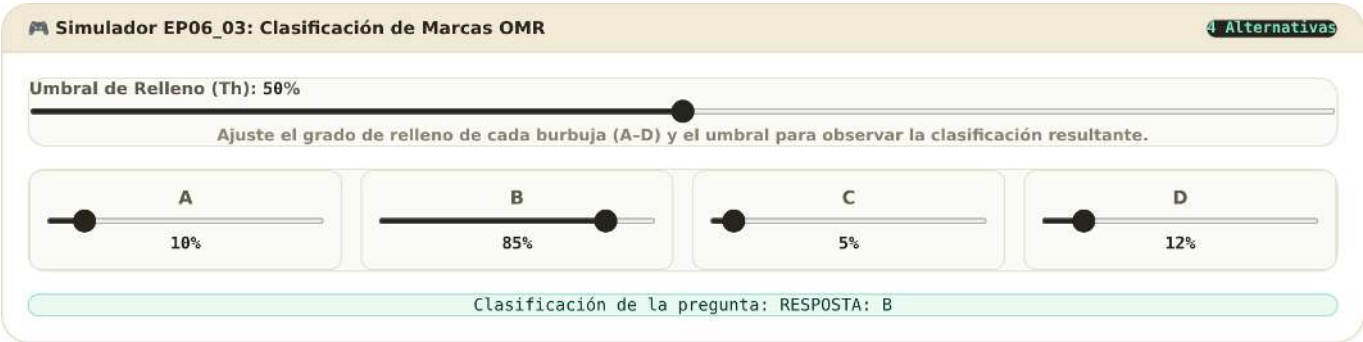
- Línea 1: Entero Q.
- Línea 2: Entero K.
- Línea 3: Número real Th.
- Siguientes Q líneas: K números reales, correspondientes a los grados de relleno de las alternativas.
- Línea 1: Entero Q y K.

Salida:

- Q líneas, cada una conteniendo la clasificación de la respectiva pregunta.
- Línea final: `OK: x BRANCO: y DUPLA_MARCACAO: z`.

6.14.3.5 📌 Ejemplos

Entrada	Salida	Observación
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	En la primera pregunta solo `B` supera el umbral; en la segunda ninguna alternativa lo supera; en la tercera, `A` y `B` exceden el umbral.
1 2 50.0 50 50	BRANCO OK: 0 BRANCO: 1 DUPLA_MARCACAO: 0	Los valores iguales al umbral no se consideran marcaciones válidas.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap06/sim-ep0603.html

Figura 6.23: Simulador EP06_03: Clasificación de Marcaciones OMR

```
%%writefile EP06_03.py
# Código Python
```

```
Overwriting EP06_03.py
```

```
TestSuite("EP06_03.py").run()
```

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

6.14.4 EP06_04 🟡 Estimador de Inclinación por Mediana Angular (*Deskew*)

Tras la detección de bordes y la aplicación de la Transformada de Hough, se obtiene un conjunto de rectas candidatas a la orientación predominante del documento. Cada recta proporciona una estimación del ángulo de inclinación, calculada por

$$\text{ángulo} = \text{rad2deg}(\theta) - 90.$$

Sin embargo, no todas las rectas corresponden a las líneas del documento: algunas resultan de ruidos, sombras u otros elementos de la imagen. En este ejercicio, implementará la etapa de estimación robusta del ángulo de inclinación, filtrando los valores plausibles y calculando su mediana.

6.14.4.1 📋 Directrices de Implementación

1. **Cantidad:** Leer el entero M, correspondiente al número de ángulos estimados.
2. **Ángulos:** Leer los M valores reales, en grados.
3. **Filtrado:** Mantener únicamente los ángulos que satisfagan **estrictamente** $-45 <$ ángulo < 45.
4. **Ausencia de candidatos:** Si ningún ángulo permanece tras el filtrado, imprimir exactamente `SEM_CORRECAO`.
5. **Mediana:** Si existen ángulos válidos:
   - si la cantidad es impar, la mediana es el elemento central de la secuencia ordenada;
   - si es par, la mediana es la media aritmética de los dos elementos centrales.
6. **Salida:** Imprimir la mediana redondeada a dos cifras decimales (redondeo estándar, *round half away from zero*, con `np.floor(img + 0.5)`).

6.14.4.2 📌 Restricciones Computacionales

- **Intervalo abierto:** los ángulos iguales a -45 o 45 no deben considerarse.
- **Precisión:** calcular la mediana utilizando los valores originales; el redondeo debe realizarse únicamente en la salida.
- **Caso vacío:** si no hay ángulos válidos, no debe calcularse ninguna mediana.

6.14.4.3 🧠 Fundamentación Teórica

Situación	Resultado
Mayoría de los ángulos concentrados en torno a la inclinación real	La mediana aproxima la orientación del documento.
Pocos ángulos discrepantes (*outliers*)	La mediana sufre poca influencia de esos valores.
Ángulos fuera del intervalo $(-45°, 45°)$	Se descartan antes del cálculo.
Ningún ángulo válido	No se aplica corrección (`SEM_CORRECAO`).

La mediana se utiliza por ser más robusta que la media en presencia de pocos valores discrepantes, produciendo una estimación más estable de la inclinación predominante del documento.

6.14.4.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

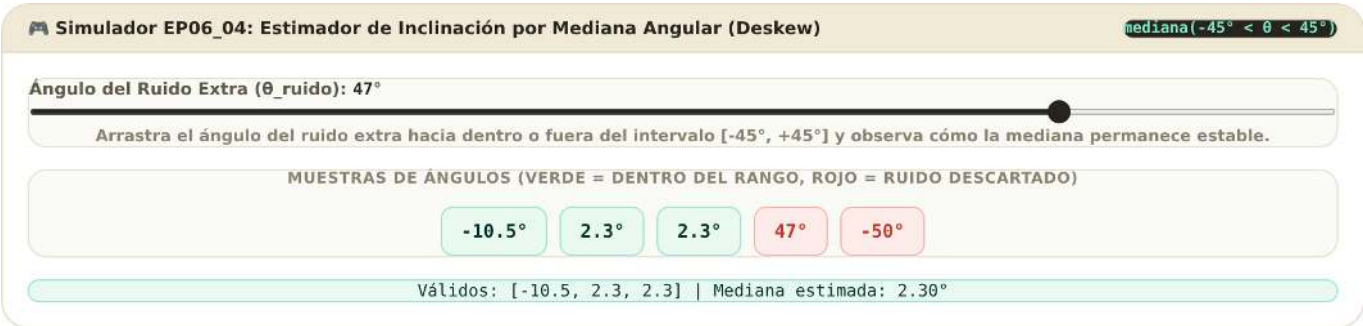
- Línea 1: Entero M.
- Línea 2: M números reales, correspondientes a los ángulos en grados.

Salida:

- Una única línea que contenga el ángulo estimado, con dos cifras decimales, o la palabra `SEM_CORRECAO` si ningún ángulo es válido.

6.14.4.5 📌 Ejemplos

Entrada	Salida	Observación
5 -50 -10.5 2.3 2.3 47	2.30	Solo se consideran los ángulos en el intervalo $(-45, 45)$; la mediana es 2,3.
4 -46 50 45 -45	SEM_CORRECAO	Ningún ángulo pertenece al intervalo abierto $(-45, 45)$.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap06/sim-ep0604.html

Figura 6.24: Simulador EP06_04: Estimador de Pendiente por Mediana Angular

```
%%writefile EP06_04.py
# Código Python
```

```
Overwriting EP06_04.py
```

```
TestSuite("EP06_04.py").run()
```

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

6.14.5 EP06_05 🟠 Normalización de Fondo por División (Corrección de Iluminación)

Un formulario fue fotografiado bajo iluminación no uniforme, lo que hace que un lado de la hoja aparezca más claro que el otro. En estas condiciones, la umbralización global por Otsu puede producir resultados insatisfactorios, ya que un único umbral no separa adecuadamente el texto y el fondo en toda la imagen. La solución presentada en el capítulo consiste en **normalizar el fondo**, dividiendo la imagen original por una versión fuertemente suavizada de sí misma, que representa la iluminación de baja frecuencia.

En este ejercicio, la imagen original y el fondo suavizado (equivalente al resultado de un `cv2.GaussianBlur` con σ elevado) ya son proporcionados. Su tarea es implementar la etapa de normalización que produce la imagen corregida.

6.14.5.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros L (filas) y C (columnas).
2. **Imagen original:** Leer los $L \times C$ valores enteros de la matriz `img` (intensidades entre 0 y 255).
3. **Fondo estimado:** Leer los $L \times C$ valores enteros de la matriz `bg` (intensidades entre 0 y 255, siempre estrictamente mayores que cero).
4. **Normalización:** Para cada posición (i, j), calcular

$$\text{valor}(i, j) = \frac{\text{img}(i, j)}{\text{bg}(i, j)} \times 255.$$

5. **Redondeo:** Redondear el resultado al entero más cercano (*round half away from zero*, con `np.floor(img + 0.5)`).
6. **Saturación:** Limitar el valor obtenido al intervalo $[0, 255]$.
7. **Salida:** Imprimir la matriz `img_norm` resultante.

6.14.5.2 📌 Restricciones Computacionales

- **División por cero:** la entrada garantiza $\text{bg}(i, j) > 0$ en todas las posiciones.
- **Orden de las operaciones:** primero redondear, luego aplicar la saturación.
- **Procesamiento independiente:** cada píxel debe normalizarse individualmente, sin utilizar información de los píxeles vecinos.

6.14.5.3 🧠 Fundamentación Teórica

Situación	Efecto de la normalización
$\text{img}(i,j) = \text{bg}(i,j)$	Resultado igual a 255, correspondiente al fondo normalizado.
$\text{img}(i,j) < \text{bg}(i,j)$	Resultado menor que 255, preservando regiones más oscuras, como el texto.
$\text{img}(i,j) > \text{bg}(i,j)$	Resultado superior a 255, posteriormente saturado.
Fondo con iluminación no uniforme	La división reduce las variaciones lentas de iluminación, haciendo la imagen más homogénea.

La división por el fondo estimado reduce los efectos de la iluminación no uniforme y preserva el contraste entre el primer plano y el fondo, facilitando las etapas posteriores de segmentación.

6.14.5.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Siguientes L líneas: elementos de la matriz `img`.
- Siguientes L líneas: elementos de la matriz `bg`.

Salida:

- Matriz `img_norm`, con L filas y C columnas, conteniendo valores enteros separados por espacios.

6.14.5.5 📌 Ejemplos

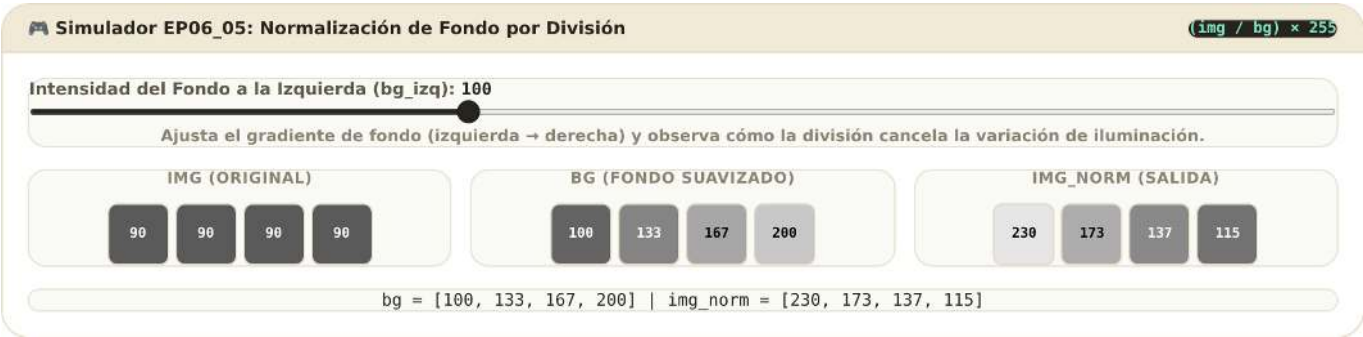
Entrada	Salida	Observación
2 2 60 120 180 40 100 100 200 80	153 255 230 128	Los valores superiores a 255 deben saturarse; $180/200 \times 255 = 229{,}5$ resulta en 230 después del redondeo.
1 3 30 60 90 60 60 60	128 255 255	Solo el primer valor permanece por debajo de 255 después de la normalización.

```
%%writefile EP06_05.py
# Código Python
```

```
Overwriting EP06_05.py
```

```
TestSuite("EP06_05.py").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap06/sim-ep0605.html

Figura 6.25: Simulador EP06_05: Normalización de Fondo por División

6.14.6 EP06_06 🔴 Mapa de Varianza Local para Detección de Textura

Una fábrica de tejidos necesita inspeccionar rollos de tela en tiempo real, sin disponer de una imagen de referencia — cada rollo presenta pequeñas variaciones naturales. En esta situación, la estrategia presentada en el capítulo consiste en analizar la **homogeneidad local de la textura**: las regiones uniformes presentan baja varianza de intensidad en pequeñas vecindades, mientras que rasguños, manchas y fallas de fabricación producen aumentos locales de dicha varianza.

En este ejercicio, implementará el núcleo de ese método, calculando la varianza local en una ventana deslizante y generando una máscara binaria que identifica las regiones cuya varianza supera un umbral.

6.14.6.1 📋 Directrices de Implementación

1. **Dimensiones y parámetros:** Leer los enteros L, C, k (tamaño de la ventana, siempre impar) y T (umbral de varianza).
2. **Imagen:** Leer los $L \times C$ valores enteros de la matriz de textura (intensidades entre 0 y 255).
3. **Tratamiento de bordes:** Cuando la ventana sobrepase los límites de la imagen, utilizar **replicación de borde**, es decir, repetir el valor del píxel válido más cercano.
4. **Media local:** Para cada posición (i, j), calcular
$$\mu(i,j) = \frac{1}{k^2} \sum_{(p,q)\in\text{ventana}} \text{textura}(p,q).$$
5. **Varianza local:** Calcular la varianza poblacional de la ventana,
$$\sigma^2(i,j) = \frac{1}{k^2} \sum_{(p,q)\in\text{ventana}} \left(\text{textura}(p,q) - \mu(i,j)\right)^2,$$
o, de forma equivalente,
$$\sigma^2(i,j) = \overline{x^2} - \mu(i,j)^2,$$
donde $\overline{x^2}$ representa la media de los cuadrados de las intensidades.
6. **Redondeo:** Redondear la varianza al entero más cercano (*round half away from zero*, con `np.floor(res_norm + 0.5)`).
7. **Umbralización:** Definir máscara$(i, j) = 1$ si la varianza redondeada es **estrictamente mayor** que T; de lo contrario, definir máscara$(i, j) = 0$.
8. **Salida:** Imprimir la máscara binaria resultante.

6.14.6.2 📌 Restricciones Computacionales

- **Replicación de borde:** utilizar el valor del píxel válido más cercano siempre que la ventana sobrepase los límites de la imagen.
- **Varianza poblacional:** utilizar denominador k^2, nunca $k^2 - 1$.
- **Comparación estricta:** la máscara debe calcularse utilizando la condición $\sigma^2_{\text{redondeada}} > T$.
- **Ventana impar:** el valor de k es siempre impar, garantizando un píxel central.

6.14.6.3 🧠 Fundamentación Teórica

Situación	Varianza local	Interpretación
Región uniforme	Baja	Intensidades similares en la vecindad.
Región que contiene un defecto	Alta	La presencia de intensidades distintas aumenta la dispersión de los valores.
Ventana pequeña	Mayor sensibilidad a detalles y ruido	Detecta alteraciones localizadas.
Ventana grande	Respuesta más suave	Evidencia defectos mayores, pero reduce la precisión de su localización.

La varianza local mide la dispersión de las intensidades en una vecindad. Las regiones homogéneas presentan baja varianza, mientras que las alteraciones en la textura aumentan esta medida, permitiendo identificar posibles defectos mediante una simple umbralización.

6.14.6.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Entero k (impar).
- Línea 4: Entero T.
- Siguientes L líneas: elementos enteros de la matriz de textura.

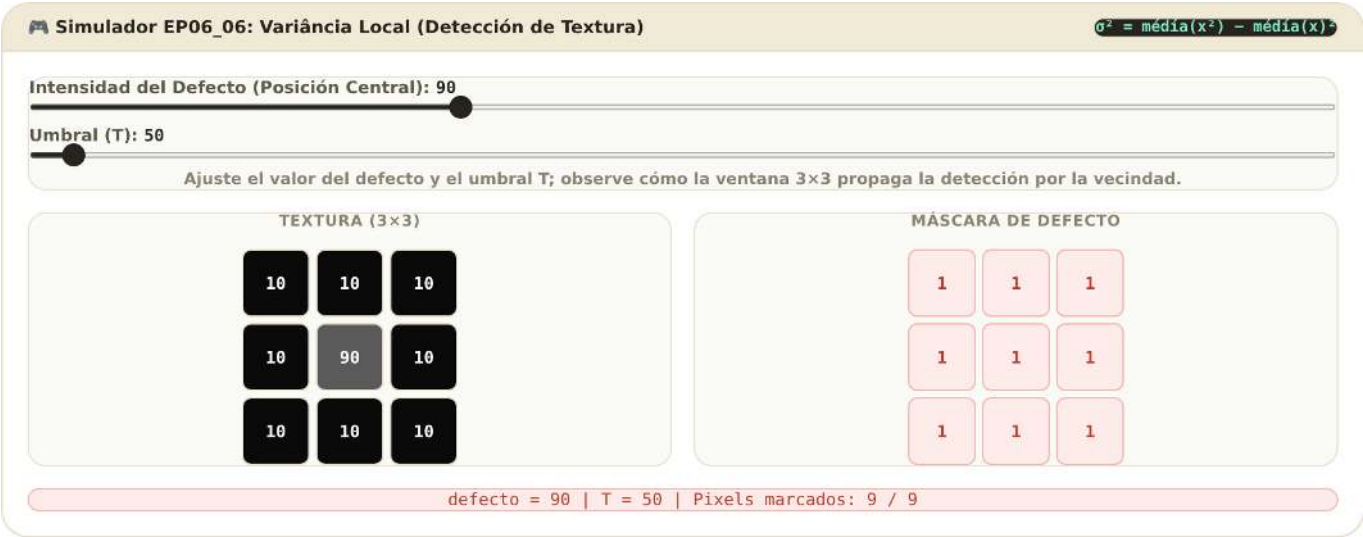
Salida:

- Máscara binaria (valores 0 o 1), con L filas y C columnas.

6.14.6.5 📌 Ejemplos

Entrada	Salida	Observación
3 3 3 50 10 10 10 10 10 10 10 90 10	0 0 0 1 1 1 1 1 1	El defecto aumenta la varianza en todas las ventanas que lo contienen.

Entrada	Salida	Observación
2 2 3 5 100 100 100 100	0 0 0 0	La textura es uniforme; la varianza es nula en toda la imagen.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap06/sim-ep0606.html

Figura 6.26: Simulador EP06_06: Mapa de Variância Local para Detecção de Textura

```
%%writefile EP06_06.py
# Código Python
```

```
Overwriting EP06_06.py
```

```
TestSuite("EP06_06.py").run()
```

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

6.14.7 EP06_07 🟣 *Pipeline* de Inspección Industrial: Registro por Traslación y Sustracción

En una línea de producción, una cámara fija fotografía cada pieza que pasa por la cinta transportadora, comparándola con una imagen de referencia sin defectos. El problema: pequeñas vibraciones de la cinta desplazan la pieza en relación con la posición de referencia en cada captura. Si la sustracción de imágenes se aplica directamente, sin corrección, el desplazamiento por sí solo ya genera diferencias enormes — **falsos positivos** que enmascaran los defectos reales.

Este es el ejercicio más completo del capítulo: debes **primero registrar** (alinear geométricamente) la imagen capturada usando un desplazamiento conocido (dx, dy), proporcionado por un sensor de posición de la cinta, y **solo entonces aplicar la sustracción** con umbralización, exactamente como se describe en la sección de inspección industrial.

6.14.7.1 📋 Directrices de Implementación

1. **Dimensiones y parámetros:** Leer L, C (dimensiones de las imágenes), el desplazamiento entero conocido dx, dy (pudiendo ser negativos) y el umbral de detección T (entero).

2. **Imágenes:** Leer la matriz de referencia (`ref`, $L \times C$, sin defectos) y la matriz capturada (`cap`, $L \times C$, posiblemente desplazada y con defecto).
3. **Registro por traslación:** Construir la imagen alineada `alin` aplicando el desplazamiento (dx, dy) recibido:
$$\text{alin}(i,j) = \begin{cases} \text{cap}(i+dy,\ j+dx), & \text{si } (i+dy,\ j+dx) \in [0,L) \times [0,C) \\ 0, & \text{en caso contrario} \end{cases}$$
4. **Relleno de borde:** Las posiciones que "salen" de la imagen capturada después del desplazamiento reciben el valor **0** (*zero-padding* — fuera del campo de visión de la cámara; **nota que este ejercicio usa cero, diferente de la replicación de borde del EP06_06**).
5. **Diferencia absoluta:** Calcular, píxel a píxel,
$$\text{diff}(i,j) = |\text{ref}(i,j) - \text{alin}(i,j)|$$
6. **Umbralización:** Definir $\text{máscara}(i,j) = 1$ si $\text{diff}(i,j) > T$; en caso contrario, $\text{máscara}(i,j) = 0$.
7. **Salida:** En este orden — (a) la matriz `alin` ($L \times C$); (b) la máscara de defecto ($L \times C$); (c) una última línea con el total de píxeles clasificados como defectuosos.

6.14.7.2 📌 Restricciones Computacionales

- ***Zero-padding*, no replicación:** posiciones fuera de los límites de la imagen capturada, después del desplazamiento, valen exactamente 0 — este es el punto que más diferencia este ejercicio del EP06_06.
- **Comparación estricta:** $\text{diff}(i,j) > T$.
- **Signo de (dx, dy):** el desplazamiento puede ser positivo o negativo; la fórmula del paso 3 debe aplicarse literalmente, sin invertir los signos.
- **Todos los valores son enteros:** no hay redondeo en esta etapa.

6.14.7.3 🧠 Fundamentación Teórica

Etapa omitida	Consecuencia
Omitir el registro geométrico	Todo el borde de la imagen (introducido por el desplazamiento) se marca como "defecto" — falso positivo sistemático
Registro con (dx, dy) incorrecto	Pieza y referencia quedan desalineadas; la sustracción detecta contornos desplazados, no defectos reales
Umbral T demasiado bajo	Ruido de captura (variaciones de 1–2 niveles de gris) se confunde con defecto
Umbral T demasiado alto	Defectos sutiles dejan de ser detectados

El registro geométrico y la sustracción son etapas complementarias: el primero garantiza que ambas imágenes representen exactamente la misma escena en el mismo referencial espacial; la segunda aísla lo que realmente cambió entre ellas — idealmente, solo los defectos.

6.14.7.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

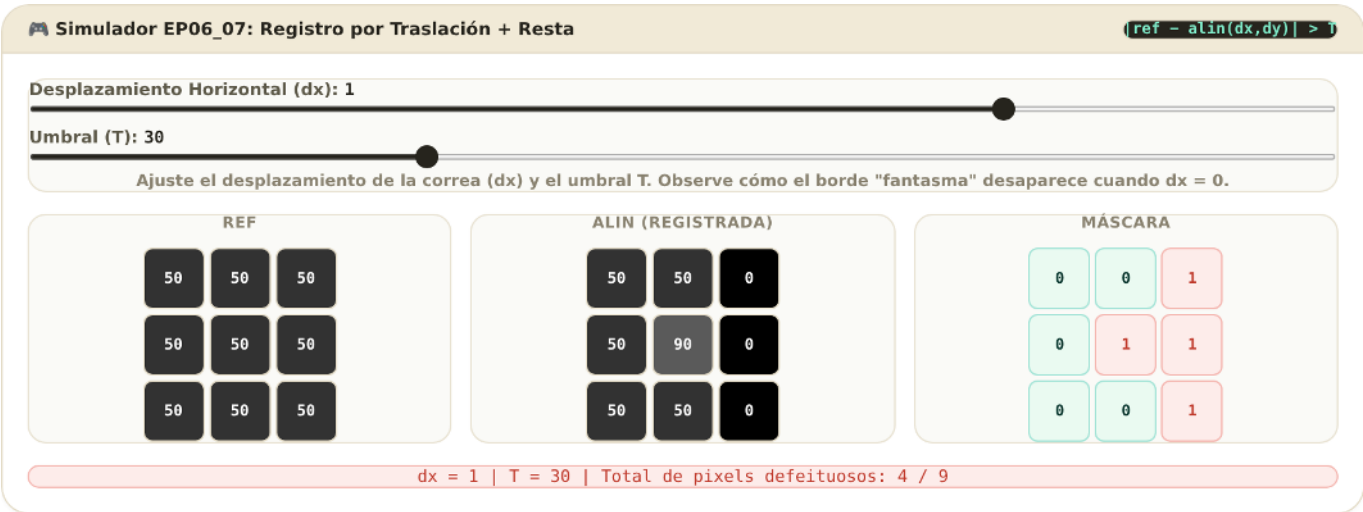
- Línea 1: Entero L.
- Línea 2: Entero C.
- Línea 3: Dos enteros dx y dy, separados por espacio.
- Línea 4: Entero T.
- Siguientes L líneas: elementos enteros de la matriz `ref`.
- Siguientes L líneas: elementos enteros de la matriz `cap`.

Salida:

- L líneas con la matriz `alin`.
- L líneas con la máscara de defecto (0/1).
- Última línea: `Total de píxeles defectuosos: X`.

6.14.7.5 📌 Ejemplos

Entrada	Salida	Observación
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 píxeles defectuosos: 4	$dx = 1$ desplaza la lectura una columna a la derecha; la última columna de `alin` queda sin correspondencia (se convierte en 0) y se marca sistemáticamente; el defecto real (90) también se detecta.
2 2 0 0 20 10 10 10 10 10 10 10 60	10 10 10 60 0 0 0 1 Total de píxeles defectuosos: 1	Sin desplazamiento ($dx = dy = 0$): `alin` es idéntica a `cap`; solo el defecto real (60) se detecta.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap06/sim-ep0607.html

Figura 6.27: Simulador EP06_07: *Pipeline* de Inspección — Registro por Traslación y Sustracción

```
%%writefile EP06_07.py
# Código Python
```

```
Overwriting EP06_07.py
```

```
TestSuite("EP06_07.py").run()
```

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

6.14.8 EP06_08 ⚫ Segmentación y Decodificación Real de *QRCode* con OpenCV

En los ejercicios anteriores, las magnitudes intermedias del *pipeline* de procesamiento de imágenes — como áreas, perímetros, varianzas y desplazamientos — se proporcionaron directamente o se calcularon a partir de matrices numéricas, sin necesidad de bibliotecas especializadas de Visión por Computador. En este ejercicio de cierre del capítulo, esta restricción se elimina de forma intencional: se utilizará la biblioteca **OpenCV** (`cv2`) para localizar y decodificar un *QRCode* real presente en una escena.

La propuesta reproduce un flujo simplificado de sistemas empleados en inspección visual, automatización industrial y lectura automática de documentos. Para mantener la entrada de datos accesible al contexto educativo, la carga de la imagen se integrará a la biblioteca didáctica `morph`, mediante la función `mm.readImg`.

La escena se proporciona en el formato **PGM ASCII (P2)** y contiene un único *QRCode* válido, además de diversos **objetos distractores**, como rectángulos, regiones de ruido texturizado y bloques aislados. La segmentación basada únicamente en propiedades geométricas — como área y forma aproximadamente cuadrada — es necesaria para reducir el espacio de búsqueda, pero no es suficiente para identificar el código correcto. La confirmación final se realizará exclusivamente mediante el intento de decodificación utilizando `cv2.QRCodeDetector`, procedimiento compatible con aplicaciones reales de reconocimiento automático.

6.14.8.1 📋 Directrices de Implementación

1. **Lectura de las dimensiones y parámetros**

   Leer, en este orden, mediante la entrada estándar:

   - una línea que contenga el número de filas L;
   - una línea que contenga el número de columnas C;
   - una línea que contenga los cuatro parámetros del algoritmo separados por espacios:
     - umbral de binarización T (entero);
     - área mínima $A_{\min}$ (entero);
     - tolerancia de aspecto tol (real);
     - margen M (entero, en píxeles).

2. **Carga de la imagen**

   Utilizar la función didáctica `f = mm.readImg(L, C)` para leer los $L \times C$ valores de la imagen en tonos de gris, obteniendo un *array* de NumPy de tipo `uint8`.

3. **Binarización**

   Aplicar umbralización binaria invertida utilizando el umbral T. Todo píxel de la imagen original con intensidad estrictamente mayor que T debe convertirse a 255, mientras que los demás deben asumir el valor 0.

4. **Detección de contornos**

   Extraer los componentes conectados externos utilizando `cv2.findContours(...)` con los parámetros:

   - `cv2.RETR_EXTERNAL`;
   - `cv2.CHAIN_APPROX_SIMPLE`.

5. **Filtrado geométrico**

   Para cada contorno encontrado:

   - calcular el rectángulo delimitador `(x, y, w, h)` mediante `cv2.boundingRect`;
   - mantener únicamente los candidatos que satisfagan simultáneamente:

     Área mínima

$$w \times h > A_{\min}$$

Razón de aspecto

$$\left|\frac{w}{h} - 1\right| \leq \text{tol}$$

6. **Ordenación de los candidatos**

   Ordenar los candidatos por el área del rectángulo delimitador

   $$w \times h$$

   en orden descendente.

   En caso de empate, preservar el orden originalmente devuelto por `cv2.findContours`.

7. **Verificación por decodificación**

   Para cada candidato, siguiendo el orden establecido:

   - expandir el rectángulo en M píxeles en las cuatro direcciones;
   - limitar los índices para permanecer dentro de la imagen;
   - extraer el recorte directamente de la imagen original `f`;
   - aplicar `cv2.QRCodeDetector().detectAndDecode(...)` sobre ese recorte.

8. **Criterio de detención**

   Interrumpir inmediatamente el procesamiento cuando el primer candidato produzca una *cadena* decodificada no vacía.

9. **Caso no encontrado**

   Si ningún candidato se decodifica con éxito, imprimir exactamente: `QRCODE_NAO_ENCONTRADO`

10. **Salida (caso encontrado)**

    Imprimir dos líneas.

    Primera línea: `linha coluna altura largura` utilizando el rectángulo delimitador **original**, antes de la expansión por el margen M.

    Segunda línea: `texto_decodificado`

6.14.8.2 📌 Restricciones Computacionales

- Utilizar funciones de OpenCV para realizar la binarización, la detección de contornos, el cálculo del rectángulo delimitador y la decodificación del QRCode.
- El filtrado geométrico debe ocurrir obligatoriamente antes de la etapa de decodificación.
- Utilizar exclusivamente el umbral fijo T proporcionado en la entrada. No se permite utilizar métodos automáticos de umbralización, como Otsu o umbralización adaptativa.
- Garantizar que los recortes enviados al decodificador permanezcan dentro de los límites de la imagen.

6.14.8.3 🧠 Fundamentación Teórica

Etapa	Papel en el pipeline	Consecuencia si se omite
Filtrado geométrico	Reduce el espacio de búsqueda seleccionando solo regiones compatibles con la geometría esperada de un QRCode.	El decodificador procesaría todos los contornos, incluyendo ruidos y objetos distractores.
Decodificación	Confirma semánticamente si el candidato contiene un QRCode válido.	Objetos geométricamente similares podrían clasificarse incorrectamente como QRCode.
Margen M	Preserva la *zona de silencio* alrededor del código, facilitando su detección.	La ausencia de este margen puede impedir la alineación y la lectura correcta del código.

Este ejercicio integra conceptos estudiados a lo largo del capítulo en un único *pipeline* de Visión por Computador. La segmentación reduce el conjunto de regiones candidatas mediante características geométricas, mientras que la etapa de decodificación valida el contenido de la región utilizando un algoritmo especializado de reconocimiento.

6.14.8.4 📦 Especificación de Entrada y Salida (VPL)

Estructura de Entrada

```
L
C
T A_min tol M
[matriz de la imagen]
```

Estructura de Salida (Éxito)

```
linha coluna altura largura
texto_decodificado
```

Estructura de Salida (Fallo)

```
QRCODE_NAO_ENCONTRADO
```

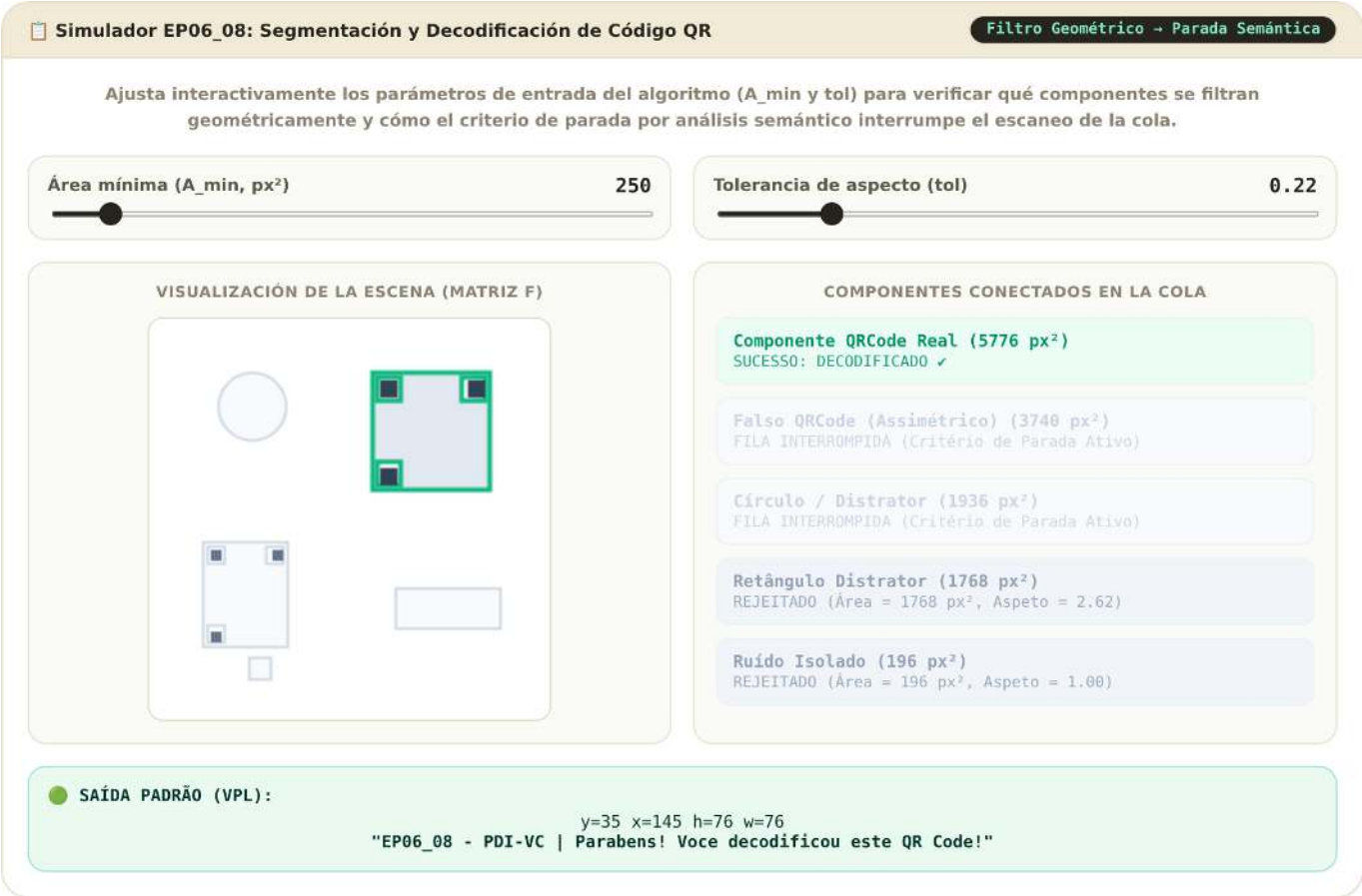
6.14.8.5 📌 Archivos de Referencia (.pgm)

Para fines de validación, depuración local y análisis de matrices reales de píxeles, los archivos de imagen generados en el estándar ASCII P2 se encuentran disponibles en el directorio del proyecto. Puede utilizarlos para probar con decodificadores de su teléfono móvil la adherencia de su código (guardar *.pgm localmente para visualizar):

- 📥 **Caso 1: Patrón Normal** – Contiene un único código perfectamente centrado con distractores geométricos simples en la periferia.
- 📥 **Caso 2: Escenario Complejo** – Presenta mayor densidad de ruido texturizado y múltiples distractores candidatos que ponen a prueba los límites del filtrado por aspecto.
- 📥 **Caso 3: Mensaje Expandido** – Contiene un QRCode estructurado a partir de una cadena de caracteres de mayor longitud, generando una mayor densidad de módulos internos.
- 📥 **Caso 4: Geometría Compacta** – Evalúa el comportamiento del pipeline bajo condiciones optimizadas de contraste y posicionamiento límite.
- 📥 **Caso 5: Escenario de Exclusión** – Imagen compuesta puramente por elementos distractores de alta área, diseñada para validar el comportamiento de fallo controlado del programa.

```
%%writefile EP06_08.py
# Código Python
```

```
Overwriting EP06_08.py
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap06/sim-ep0608.html

Figura 6.28: Simulador EP06_08: Segmentación Geométrica + Verificación por Decodificación de Código QR

```
TestSuite("EP06_08.py").run()
```

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

Capítulo 7

Clasificación de Imágenes y Reconocimiento de Patrones

Executar Colab Abrir GitHub

En el **Capítulo 6**, la transición de la Parte I a la Parte II se presentó mediante dos aplicaciones que ya requerían decisiones automatizadas: el reconocimiento de marcas en hojas de respuesta (OMR) y la detección de defectos en inspección industrial. En ambos casos, sin embargo, las decisiones dependían de reglas geométricas y de umbrales definidos manualmente, como determinar si un disco era suficientemente circular o si una región era lo bastante oscura.

Este capítulo formaliza el problema más general subyacente a estas aplicaciones: dado un conjunto de ejemplos etiquetados, ¿cómo entrenar un sistema para **clasificar automáticamente** nuevas imágenes o regiones de interés? Esta cuestión se encuentra en el núcleo del **Reconocimiento de Patrones**, disciplina que fundamenta gran parte de las tareas modernas de Visión Computacional, desde la clasificación de imágenes hasta la detección de objetos y la segmentación semántica, exploradas en los próximos capítulos.

Se estudiarán los principales **descriptores clásicos de imagen** (color, textura y forma/gradiente) y el clasificador **k-Vecinos más Cercanos** (*k-Nearest Neighbors* — k-NN), elegido por su simplicidad conceptual y por evidenciar, de forma directa, la relación entre el espacio de características, las métricas de distancia y las fronteras de decisión — conceptos que permanecen centrales incluso en clasificadores basados en redes neuronales profundas, estudiados en el capítulo final de esta parte.

7.1 Objetivos del Capítulo

Al final de este capítulo, el estudiante deberá ser capaz de:

- **Comprender el *pipeline* clásico de reconocimiento de patrones**: adquisición, preprocesamiento, extracción de descriptores, clasificación y evaluación;
- **Extraer e interpretar descriptores clásicos** de color, textura (*Local Binary Patterns* — LBP) y forma/gradiente (*Histogram of Oriented Gradients* — HOG);
- **Implementar y entrenar un clasificador k-NN** para tareas de clasificación de imágenes;
- **Evaluar clasificadores** mediante métricas como exactitud, matriz de confusión, precisión y recuperación;
- **Analizar el efecto del parámetro k** y de la dimensionalidad del espacio de características en el rendimiento del clasificador;
- **Reconocer las limitaciones de los descriptores artesanales** (*hand-crafted features*) y comprender la motivación para la transición, en los próximos capítulos, hacia descriptores aprendidos automáticamente.

7.2 Configuración del Entorno

Los ejemplos de este capítulo utilizan bibliotecas ampliamente empleadas en Procesamiento Digital de Imágenes, Visión por Computadora y Aprendizaje Automático. El bloque siguiente instala los paquetes

necesarios; en entornos que ya los tengan, la ejecución puede omitirse.

```
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 importlib
import subprocess
import sys

def setup_cap07():
    """Instala bibliotecas ausentes necesarias para este capítulo
    (visión computacional y aprendizaje automático)."""
    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"[AVISO] Error al instalar {pacote} (necesario para el módulo
                {modulo}).")


setup_cap07()

# ==========================================================
# Bibliotecas
# ==========================================================

# Computación científica
import numpy as np
import pandas as pd

# Visualización
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import seaborn as sns

# Visión computacional
import cv2
from skimage import data as skdata
from skimage.feature import hog, local_binary_pattern

# Aprendizaje automático
```

```
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,
)
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0
```

7.3 Un Problema Concreto: Clasificación de Frutas

Antes de presentar los fundamentos teóricos, considere el siguiente problema, que se utilizará como ejemplo a lo largo de este capítulo para ilustrar los principales conceptos del reconocimiento de patrones.

El escenario: Una granja automatizada utiliza un sistema de Visión Computacional para separar manzanas, plátanos y naranjas en líneas de envasado.

El desafío: Las frutas llegan a la cinta en diferentes posiciones y orientaciones, bajo condiciones de iluminación que pueden variar. Además, las hojas, las sombras y las pequeñas oclusiones pueden dificultar su identificación. ¿Cómo desarrollar un sistema capaz de clasificarlas correctamente?

Un posible enfoque:

1. Extraer descriptores que representen características relevantes de las frutas:
   - **Color:** distribución predominante de los colores;
   - **Textura:** diferencias en la superficie de la cáscara;
   - **Forma:** características geométricas del contorno.
2. Entrenar un clasificador utilizando ejemplos previamente etiquetados.
3. Utilizar el modelo entrenado para clasificar automáticamente nuevas frutas.

La Figura 7.1 ilustra, de forma conceptual, cómo diferentes frutas pueden representarse en un espacio de características tridimensional.

> Reflexiona antes de continuar
>
> Si cada fruta se representara únicamente por los valores de intensidad de sus píxeles, ¿sería posible distinguirlas de manera fiable? ¿Qué tipo de información podría extraerse de la imagen para facilitar esta tarea?

```
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()
```

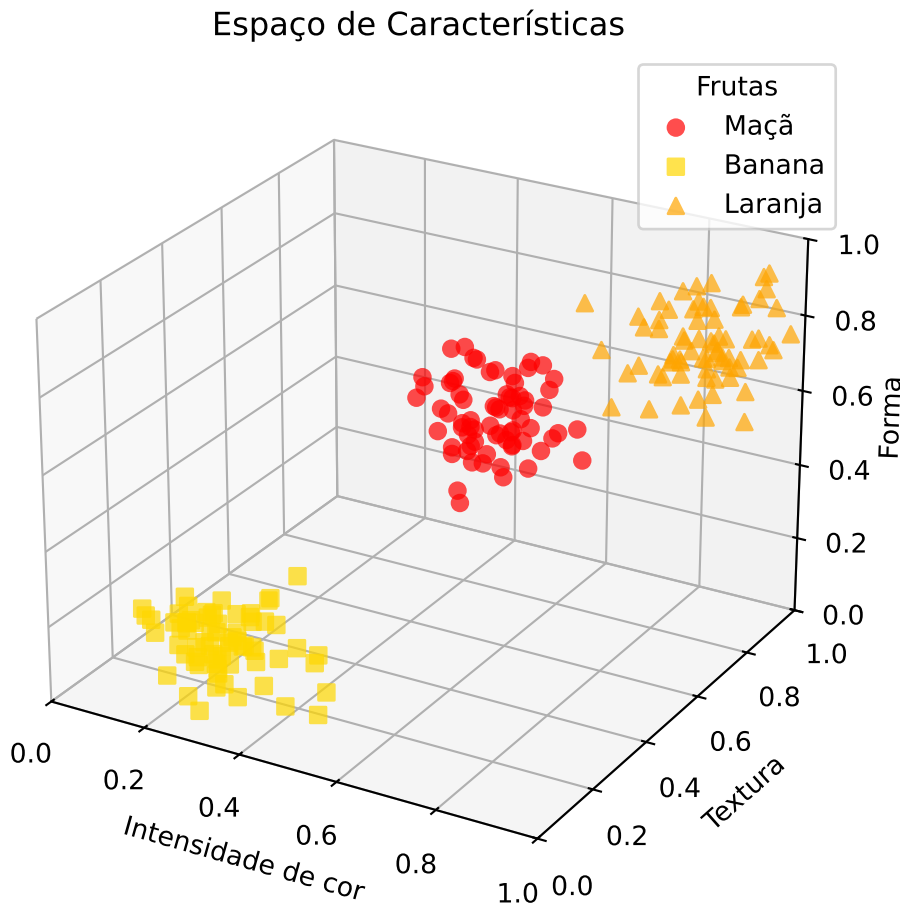


Figura 7.1: Ejemplo motivacional: diferentes frutas formando agrupaciones distintas en un espacio de características.

7.4 Panorama del Capítulo: El *Pipeline* Clásico de Clasificación de Imágenes

Antes de continuar, resulta útil presentar una visión integrada de lo que se estudiará. La Figura 8.1 muestra el flujo general de un sistema clásico de clasificación de imágenes, desde la imagen de entrada hasta la etapa de asignación de la etiqueta final. En las próximas secciones, cada una de las etapas de este proceso se estudiará en detalle.

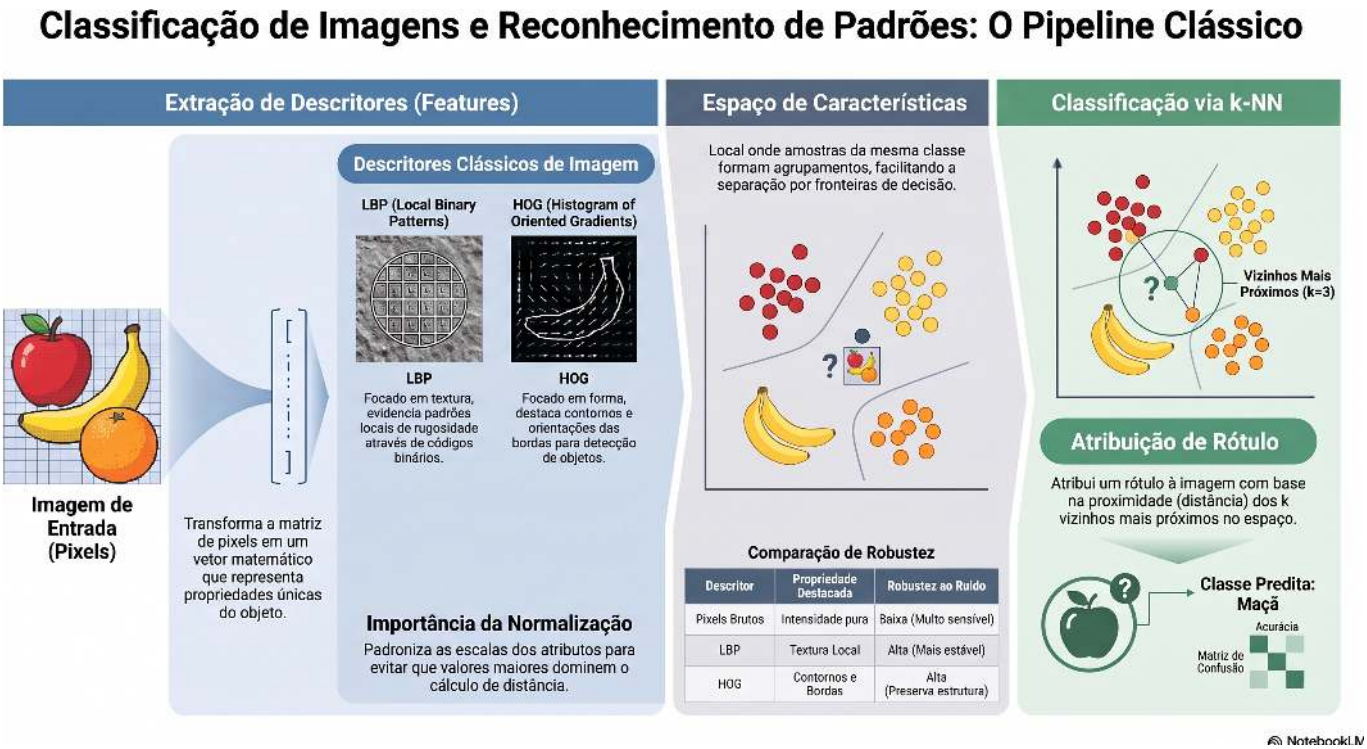


Figura 7.2: Panorama del *pipeline* clásico de clasificación de imágenes: extracción de descriptores (LBP, HOG), formación del espacio de características y clasificación mediante k-NN. No se aplica a modelos de Aprendizaje Profundo (CNNs, YOLO), que aprenden *features* de extremo a extremo directamente de los píxeles. **Fuente:** elaborado con ayuda de Gemini Notebook ({GOOGLE}, 2025).

> **ℹ Alcance de este capítulo**
>
> En este capítulo, el enfoque recae exclusivamente sobre el clasificador k-NN, por su simplicidad didáctica y por ilustrar de forma intuitiva el concepto de espacio de características. Otros clasificadores tradicionales ampliamente utilizados en Reconocimiento de Patrones — como Árboles de Decisión, Reglas de Clasificación y Máquinas de Vectores de Soporte (SVM) — se discuten en profundidad en Quilici-gonzalez (2014), especialmente en su 2ª edición, actualmente en producción (QUILICI-GONZALEZ, 2026).

7.5 Fundamentos del Reconocimiento de Patrones

Un sistema de **reconocimiento de patrones** tiene como objetivo asignar una categoría (etiqueta) a una observación — una imagen completa, una región de interés o una señal — con base en ejemplos previamente etiquetados. De forma general, este proceso se organiza en las siguientes etapas:

1. **Adquisición:** obtención de la imagen o de la señal que se va a clasificar;
2. **Preprocesamiento:** normalización, eliminación de ruido, corrección geométrica o de iluminación — etapas ya estudiadas en los capítulos anteriores;
3. **Extracción de descriptores (*features*):** transformación de la imagen en un **vector de**

características de dimensión fija, que representa las propiedades relevantes para la tarea de clasificación;
4. **Clasificación:** aplicación de un modelo que asocia el vector de características a una clase;
5. **Evaluación:** análisis del rendimiento del modelo en un conjunto de datos independiente de aquel utilizado para el entrenamiento.

El conjunto de todos los vectores de características posibles constituye el **espacio de características** (*feature space*). Un buen descriptor produce representaciones que aproximan, en ese espacio, observaciones de la misma clase y alejan observaciones de clases distintas. Esta propiedad favorece métodos de clasificación basados en proximidad, como el **k-NN**, y también beneficia a diversos otros clasificadores.

La Figura 7.1 ilustra este concepto de forma esquemática: cada fruta se representa mediante un punto en un espacio de características de tres dimensiones (color, textura y forma). Aunque ese espacio sea solo una simplificación didáctica, muestra cómo las muestras de la misma clase tienden a formar agrupaciones, mientras que clases diferentes ocupan regiones distintas, facilitando la tarea de clasificación.

7.6 Extracción de Descriptores Clásicos

Antes de la popularización de las redes neuronales profundas, los descriptores de imagen eran, en su mayoría, diseñados manualmente por expertos (*hand-crafted features*), con base en propiedades estadísticas o geométricas conocidas. Tres familias clásicas son particularmente relevantes:

- **Descriptores de color:** histogramas de intensidad o de matiz, que capturan la distribución de los valores de color de una región, ya introducidos en el **Capítulo 3** mediante la función `mm.hist`;
- **Descriptores de textura:** capturan patrones locales de repetición, rugosidad u orientación, como el *Local Binary Patterns* (LBP), estudiado a continuación;
- **Descriptores de forma/gradiente:** describen la distribución de los bordes y de las orientaciones del gradiente, como el *Histogram of Oriented Gradients* (HOG), ampliamente empleado en la detección de personas y otros objetos.

Para comparar la información capturada por cada enfoque, la Figura 7.3 muestra cómo diferentes técnicas "ven" la misma imagen.

```
# Cargar imagen de ejemplo
imagem = skdata.camera()

# Aplicar descriptores
lbp_img = local_binary_pattern(
    imagem,
    P=8,
    R=1,
    method="uniform"
    )

# Convertir el LBP a RGB solo para facilitar la visualización
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,
)

# Exhibición estandarizada
mm.show(
```

```
    [imagem, lbp_rgb, hog_img],
    titles=[
        "Imagen Original\n(como el humano ve)",
        "LBP: Textura\n(cada color = un código LBP)",
        "HOG: Gradientes y Contornos\n(regiones claras = mayor intensidad)",
    ],
    cols=3,
    figsize=(12, 4),
)

print("Observa cómo cada descriptor resalta propiedades diferentes:")
print("• LBP: evidencia patrones locales de textura.")
print("• HOG: evidencia contornos y orientaciones de los bordes.")
print("• Imagen original: contiene solo los valores de intensidad.")
```

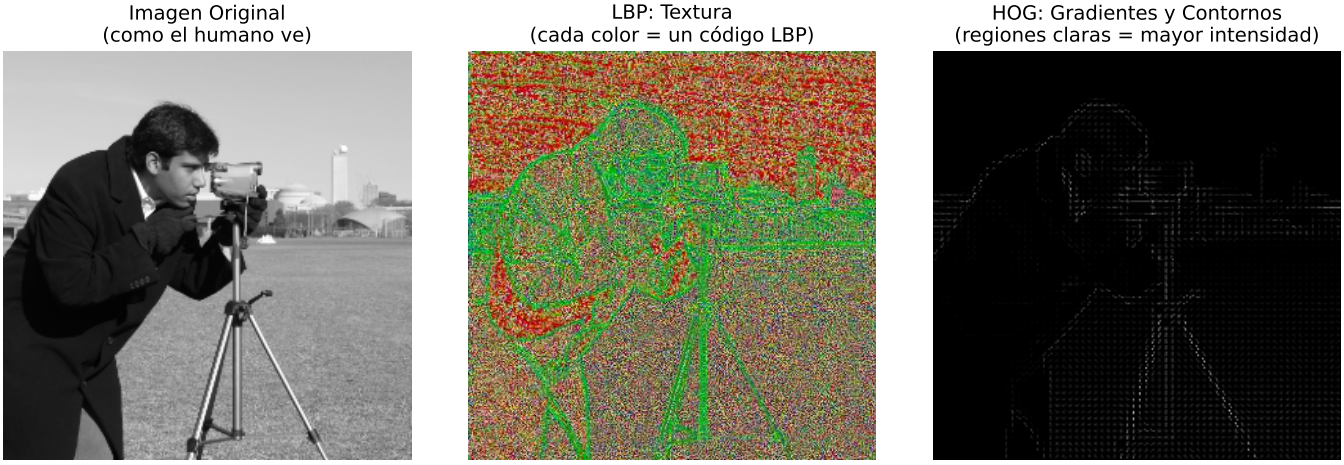


Figura 7.3: Comparación visual de diferentes descriptores aplicados a la misma imagen. Cada descriptor revela aspectos distintos de la escena.

```
Observa cómo cada descriptor resalta propiedades diferentes:
• LBP: evidencia patrones locales de textura.
• HOG: evidencia contornos y orientaciones de los bordes.
• Imagen original: contiene solo los valores de intensidad.
```

7.6.1 *Local Binary Patterns* (LBP)

El LBP es un descriptor de textura que codifica, para cada píxel central g_c, la relación entre su intensidad y la de los P vecinos dispuestos en una vecindad circular de radio R:

$$\mathrm{LBP}_{P,R}(x_c, y_c) = \sum_{p=0}^{P-1} s(g_p - g_c)\, 2^p, \qquad s(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

donde:

- (x_c, y_c) son las coordenadas del píxel central;
- g_c es la intensidad del píxel central;
- g_p es la intensidad del p-ésimo píxel vecino;
- P es el número de vecinos considerados;
- R es el radio de la vecindad circular;
- p es el índice del vecino, con $p = 0, 1, \ldots, P-1$;
- $s(z)$ es la función de umbral definida en la ecuación, donde $z = g_p - g_c$; toma el valor 1 cuando $z \geq 0$ y 0 cuando $z < 0$;

- 2^p corresponde al peso binario asociado al p-ésimo vecino.

El código LBP obtenido describe el patrón local de contraste alrededor del píxel. El histograma de estos códigos forma un vector de características compacto para representar la textura de la imagen (Figura 7.4). En este capítulo se utiliza la variante **uniforme**, que agrupa los patrones no uniformes en una única categoría, reduciendo la dimensionalidad y aumentando la robustez del 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()
```

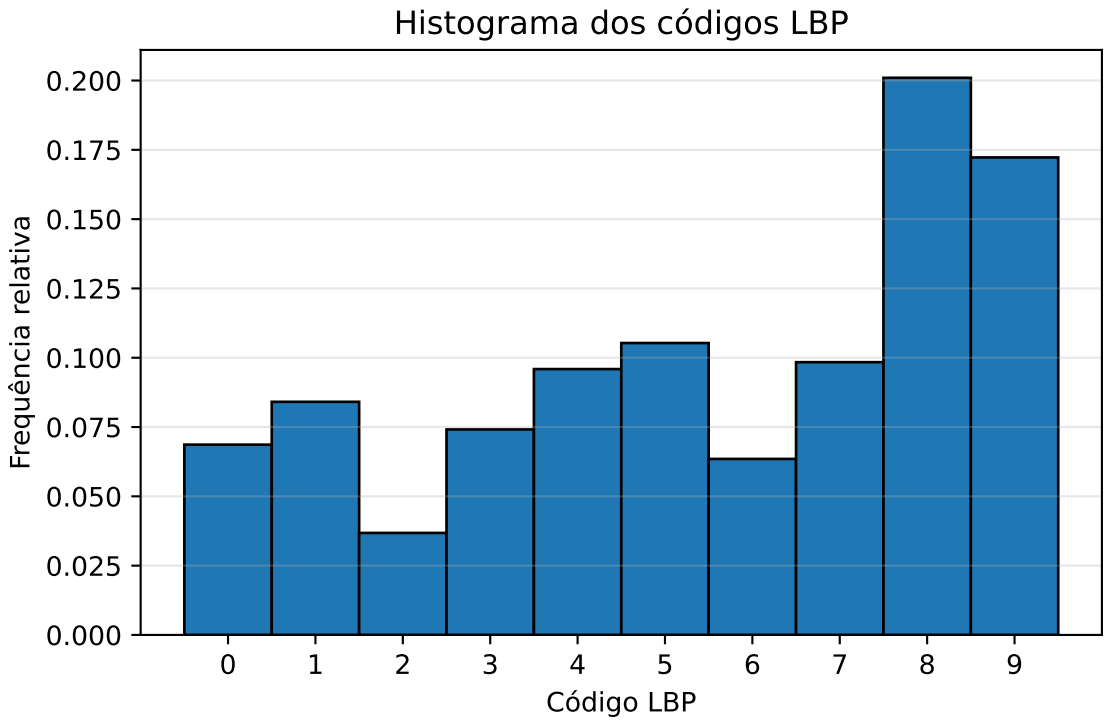


Figura 7.4: Histograma de los códigos LBP de la imagen de los camarógrafos. Cada barra representa la frecuencia relativa de un código LBP, formando el vector de características utilizado para describir la textura.

Función `local_binary_pattern`

La implementación utilizada en este capítulo es proporcionada por la biblioteca `scikit-image`:

```
local_binary_pattern(
    imagem,
    P=8,
    R=1,
    method="uniform"
)
```

donde:

- `image`: imagen en escala de grises;
- `P`: número de vecinos igualmente espaciados en la vecindad circular;
- `R`: radio de la vecindad, en píxeles;
- `method`: estrategia de codificación. En este capítulo se utiliza el valor `"uniform"`.

La ecuación presentada anteriormente describe el **LBP original**. En la implementación adoptada en este capítulo, la opción `method="uniform"` calcula inicialmente ese código y, a continuación, reasigna los patrones no uniformes a una única categoría, reduciendo la dimensionalidad del descriptor y haciéndolo más robusto ante pequeñas variaciones locales.

La Figura 7.3 presenta la representación visual del LBP, mientras que la Figura 7.4 muestra el histograma de los códigos LBP utilizado como vector de características.

El **Proyecto Práctico 2** (sección **Comparación de Descriptores para Clasificación de Texturas**) emplea el LBP en la clasificación de diferentes tipos de textura sintética.

7.6.2 *Histograma de Gradientes Orientados* (HOG)

El HOG es un descriptor que representa la forma de un objeto mediante la distribución de las orientaciones del gradiente local. Así como en el operador de Canny (**Capítulo 6**), inicialmente se calcula el gradiente:

$$|\nabla f(x,y)| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}, \qquad \theta(x,y) = \operatorname{atan2}\left(\frac{\partial f}{\partial y}, \frac{\partial f}{\partial x}\right).$$

donde:

- $f(x,y)$ es la intensidad de la imagen en el píxel (x,y);
- $\frac{\partial f}{\partial x}$ y $\frac{\partial f}{\partial y}$ son, respectivamente, las derivadas parciales de la imagen en las direcciones horizontal y vertical;
- $|\nabla f(x,y)|$ es la magnitud del vector gradiente en el píxel (x,y), que indica la intensidad de la variación local de la imagen;
- $\theta(x,y)$ es la orientación del vector gradiente en el píxel (x,y), calculada mediante la función atan2, cuyo resultado pertenece al intervalo $(-\pi,\pi]$.

Aunque $\theta(x,y)$, tal como se calcula con la función atan2, pertenece al intervalo $(-\pi,\pi]$, la implementación estándar del HOG utiliza el **gradiente no signado** (*unsigned*): las orientaciones opuestas (por ejemplo, 0 y π) se tratan como equivalentes, y los ángulos se mapean al intervalo $[0,\pi)$ antes de construir el histograma. Esta elección hace que el descriptor sea invariante a la dirección del contraste (por ejemplo, un borde claro-oscuro y un borde oscuro-claro producen la misma orientación).

La imagen se divide entonces en **celdas** (*cells*). Para cada celda, se construye un histograma de las orientaciones del gradiente, ponderado por la magnitud correspondiente. La concatenación de los histogramas de todas las celdas forma el vector de características HOG, que representa la distribución espacial de las orientaciones del gradiente y captura información sobre la forma y los contornos del objeto (Figura 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()
```

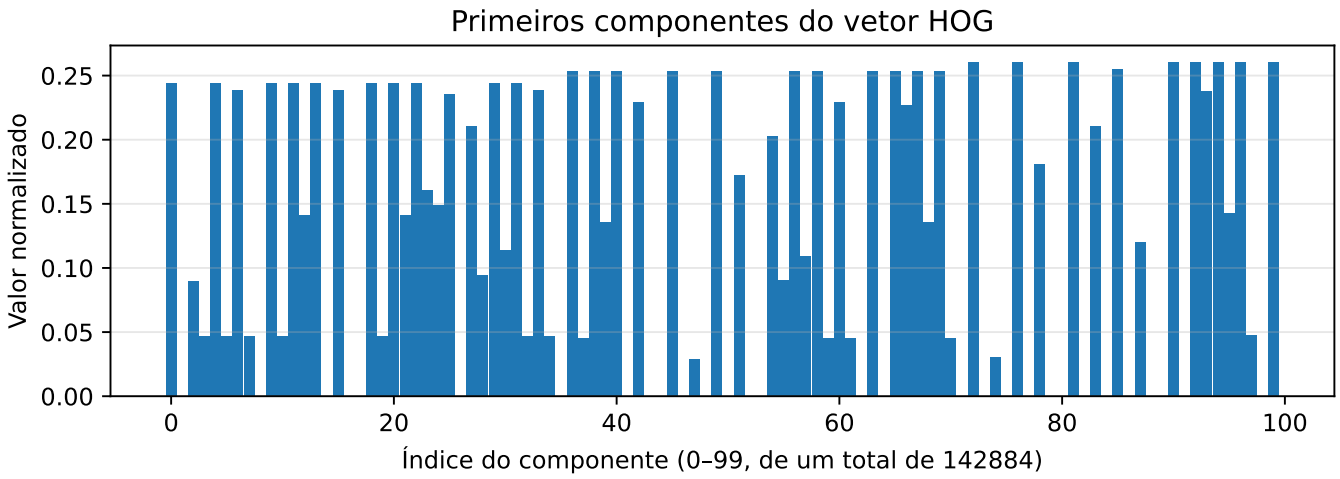


Figura 7.5: Primeros 100 componentes del vector de características HOG.

Función `hog`

La extracción del descriptor HOG se realiza mediante la función:

```
hog(
    image,
    orientations=9,
    pixels_per_cell=(8, 8),
    cells_per_block=(2, 2),
    visualize=True,
)
```

Los parámetros principales son:

- `image`: imagen de entrada;
- `orientations`: número de divisiones angulares del histograma de orientaciones en cada celda;
- `pixels_per_cell`: tamaño, en píxeles, de cada celda donde se calcula el histograma;
- `cells_per_block`: número de celdas utilizadas en la normalización del descriptor;
- `visualize`: cuando es `True`, devuelve también una imagen que ilustra los gradientes utilizados por el HOG.

La ecuación presentada anteriormente describe el cálculo de la magnitud y la orientación del gradiente, que constituyen la base del descriptor HOG. En la implementación adoptada en este capítulo, la función `hog()` utiliza esta información para construir histogramas de orientaciones en cada celda de la imagen y, a continuación, realiza la normalización en bloques (`cells_per_block`), reduciendo la sensibilidad del descriptor a variaciones de iluminación y contraste.

La Figura 7.3 presenta la representación visual del HOG, mientras que la Figura 7.5 ilustra los primeros

componentes del vector de características extraído de la imagen.

El **Proyecto Práctico 1** (sección **Clasificación de Dígitos Manuscritos con k-NN**) compara el rendimiento de los descriptores HOG con el uso directo de las intensidades de los píxeles como vector de características.

7.6.3 El Impacto de la Escala y la Normalización de Características

El clasificador k-NN toma sus decisiones basándose en la distancia entre los vectores de características. Por ello, la escala de cada característica influye directamente en el resultado de la clasificación. Si una variable presenta valores mucho mayores que las demás (por ejemplo, una intensidad de color que varía de 0 a 255, mientras que un índice de circularidad varía de 0 a 1), tiende a dominar el cálculo de la distancia, reduciendo la influencia de los otros descriptores.

Para evitar este problema, se aplica una etapa de **normalización de las características**, generalmente mediante la estandarización (*Z-score standardization*). En este procedimiento, cada característica pasa a tener media igual a cero y desviación estándar igual a uno, haciendo comparables magnitudes originalmente medidas en escalas diferentes.

La estandarización se realiza mediante la transformación

$$z = \frac{x - \mu}{\sigma},$$

donde:

- x es el valor original de la característica;
- μ es la media de esa característica calculada sobre el conjunto de entrenamiento;
- σ es la desviación estándar de la característica;
- z es el valor estandarizado.

Después de esta transformación, todas las características pasan a tener media igual a cero y desviación estándar igual a uno, permitiendo que contribuyan de forma equilibrada al cálculo de las distancias.

Clase `StandardScaler`

La estandarización utilizada en este capítulo se realiza mediante la clase `StandardScaler`, de la biblioteca `scikit-learn`:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_norm = scaler.fit_transform(X)
```

donde:

- `StandardScaler()`: crea el objeto responsable de la estandarización;
- `fit_transform(X)`: calcula la media y la desviación estándar de cada característica del conjunto `X` y devuelve la matriz estandarizada.

En la práctica, el método `fit_transform()` ejecuta dos etapas: primero (`fit`), estima la media (μ) y la desviación estándar (σ) de cada característica; a continuación (`transform`), aplica la transformación de estandarización presentada anteriormente a todos los valores de la matriz de entrada.

La Figura 7.6 muestra el efecto de la normalización. Visualmente, la distribución de los puntos permanece igual; lo que cambia es la escala de los ejes. Sin la normalización, la característica de mayor magnitud domina el cálculo de las distancias entre las muestras. Tras la estandarización, todas las características pasan a contribuir de forma equilibrada al cálculo de las distancias utilizadas por el clasificador k-NN.

```
# Datos sintéticos con escalas muy diferentes
np.random.seed(42)
X_demo = np.random.randn(20, 2) * [100, 1]
y_demo = np.array([0] * 10 + [1] * 10)

print("Efecto de la normalización:")
print("  Característica 1: escala  100")
print("  Característica 2: escala  1")
print("\nSin normalización, la primera característica domina el cálculo de las distancias.")
print("Con normalización, ambas contribuyen de forma equilibrada.")
print("\nLa normalización es esencial cuando las características poseen escalas diferentes.")
fig, axes = plt.subplots(1, 2, figsize=(10, 4))

# Sin normalización
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()

# Con normalización
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()
```

```
Efecto de la normalización:
  Característica 1: escala  100
  Característica 2: escala  1

Sin normalización, la primera característica domina el cálculo de las distancias.
Con normalización, ambas contribuyen de forma equilibrada.

La normalización es esencial cuando las características poseen escalas diferentes.
```

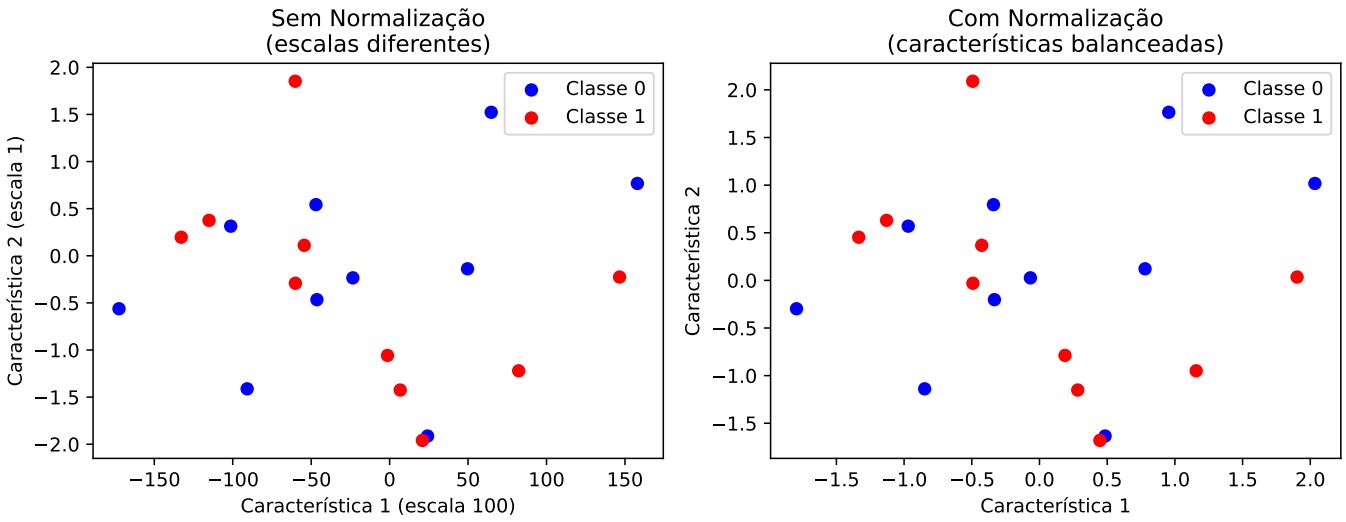


Figura 7.6: Importancia de la normalización de las características para el clasificador k-NN.

7.7 📌 Mapa Conceptual

Hasta aquí, se han presentado los descriptores clásicos (LBP, HOG, píxeles brutos) y la forma en que organizan las muestras en un espacio de características. La Figura 7.7 sintetiza ese recorrido y sitúa estas etapas dentro del flujo general de un sistema clásico de clasificación de imágenes, indicando también las etapas siguientes — clasificación (k-NN) y evaluación de los resultados — que se formalizarán en las próximas secciones.

7.8 Descriptores en la Práctica

Tras conocer los principales descriptores clásicos, es natural preguntarse cómo influyen en el rendimiento de un clasificador en situaciones cercanas a las que se encuentran en la práctica.

En esta sección, se compara el uso de tres representaciones distintas de las mismas imágenes: intensidades de los píxeles, descriptores LBP y descriptores HOG. Para hacer el experimento más realista, se añade ruido sintético a los datos, en intensidades diferentes para cada descriptor — una forma simplificada de simular el hecho de que, en la práctica, diferentes representaciones toleran de manera desigual las imperfecciones de la captura (ruido del sensor, pequeñas variaciones de posición, etc.).

La Figura 7.8 presenta las matrices de confusión obtenidas para cada descriptor, permitiendo identificar en qué clases ocurren los principales errores de clasificación. La interpretación de estas matrices fue introducida en el **Capítulo 1**, cuando se presentaron los conceptos de **Verdadero Positivo (VP)**, **Falso Positivo (FP)**, **Verdadero Negativo (VN)** y **Falso Negativo (FN)**. Estos conceptos fueron explorados en los **EPs 01_02** (métricas de clasificación) y **01_03** (*mean Average Precision* – mAP), disponibles en:

- https://fzampirolli.github.io/pdi-vc/eps/py.pt/EP01_02.html
- https://fzampirolli.github.io/pdi-vc/eps/py.pt/EP01_03.html

En este capítulo, las matrices de confusión se emplean para analizar cómo diferentes descriptores influyen en el rendimiento del clasificador.

A continuación, la Figura 7.9 resume la precisión global obtenida por cada descriptor.

Los resultados muestran que el rendimiento del clasificador depende directamente de la representación elegida para describir las imágenes. Mientras que el uso directo de las intensidades de los píxeles es más sensible a las degradaciones introducidas, los descriptores LBP y HOG preservan mejor la información relevante para la clasificación, resultando en un mayor rendimiento en este escenario. Es importante resaltar que los niveles de ruido aplicados a cada descriptor fueron elegidos solo con fines didácticos, de modo que ilustren el principio

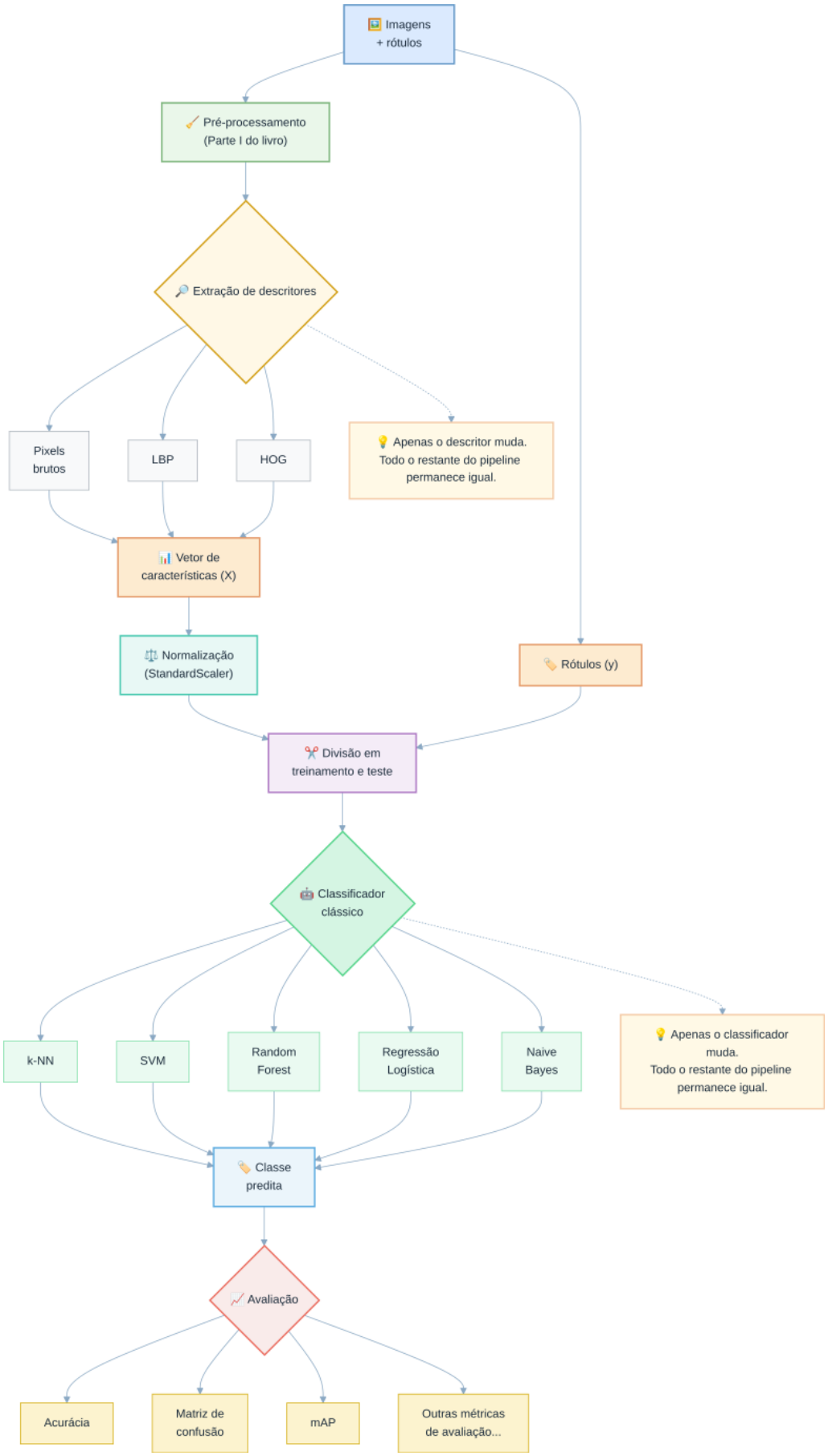


Figura 7.7: Mapa conceptual del proceso de clasificación de imágenes utilizando descriptores clásicos (LBP, HOG, píxeles brutos) y clasificador tradicional (k-NN). No se aplica a modelos de Aprendizaje Profundo (CNNs, YOLO), que aprenden *features end-to-end* directamente de los píxeles.

general de que descriptores más elaborados *pueden* ser más robustos a degradaciones — lo que no significa que esta relación se verifique siempre, como el estudio de caso de la próxima sección demostrará.

Cómo se realiza el experimento

Dado que el objetivo de esta sección es comparar únicamente el efecto de los descriptores, se genera un conjunto de datos sintético simple: tres nubes de puntos gaussianas, centradas en los mismos valores de "color, textura y forma" ya utilizados en la Figura 7.1 — el mismo patrón empleado desde el inicio del capítulo para representar las tres clases de frutas.

```
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
```

```
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)
```

donde:

- `centros`: diccionario con el punto medio de cada clase en el espacio de características (color, textura, forma);
- `n_por_classe`: número de muestras generadas por clase;
- `np.random.normal(centro, 0.12, (n_por_classe, 3))`: genera `n_por_classe` muestras alrededor de cada centro, con desviación estándar 0,12 en cada dimensión;
- `np.repeat(list(centros.keys()), n_por_classe)`: genera el vector de etiquetas correspondiente, en el mismo orden que los centros.

A continuación, se utiliza el flujo de entrenamiento y evaluación:

- `train_test_split(X, y, test_size=0.3)`: divide los datos en entrenamiento (70%) y prueba (30%);
- `KNeighborsClassifier(n_neighbors=5)`: crea un clasificador k-NN con $k = 5$ vecinos;
- `fit(X_train, y_train)`: ajusta el modelo a los datos de entrenamiento;
- `predict(X_test)`: clasifica las muestras de prueba;
- `accuracy_score(y_test, y_pred)`: calcula la precisión;
- `confusion_matrix(y_test, y_pred)`: genera la matriz de confusión.

En este experimento, el conjunto de entrenamiento, el clasificador y el método de evaluación permanecen exactamente los mismos. La única diferencia entre los experimentos es la representación utilizada para cada imagen (píxeles brutos, LBP o HOG), lo que permite evaluar exclusivamente la influencia del descriptor sobre el rendimiento del clasificador.

```
classes = ["Maçã", "Banana", "Laranja"]

np.random.seed(42)

# Mismos centros de clase (color, textura, forma) utilizados en el
# ejemplo anterior, ahora reutilizados para generar los datos
# sintéticos de entrenamiento y prueba de este experimento.
```

```
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)

# Simular descriptores con diferentes niveles de sensibilidad al ruido.
# Cuanto mayor sea el ruido añadido, peor tiende a ser la representación.
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),
}

# Crear una única figura con 3 subplots lado a lado para las 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

    # Matriz de confusión en el subplot correspondiente

    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()
```

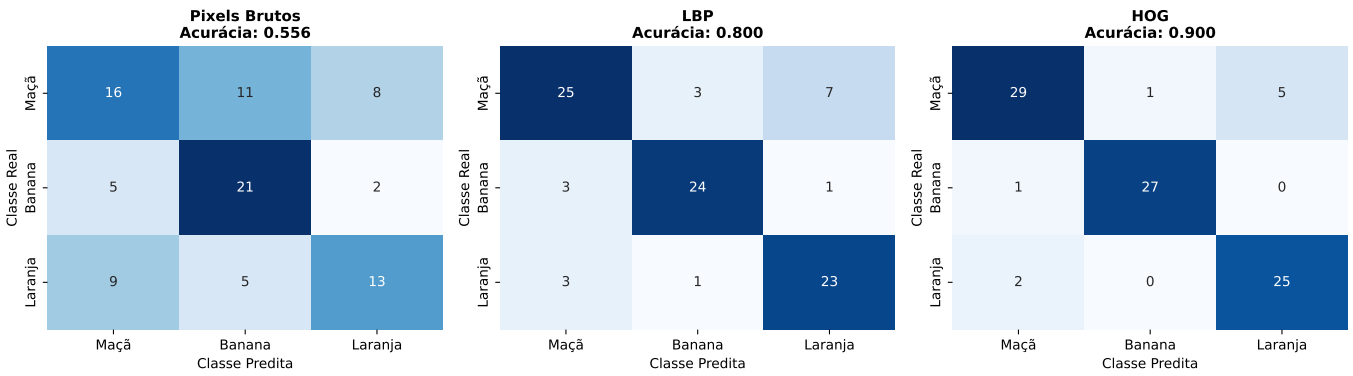


Figura 7.8: Matrices de confusión obtenidas por el clasificador k-NN utilizando tres descriptores diferentes. Las filas representan la clase real (Manzana, Plátano y Naranja) y las columnas la clase predicha. Cuanto mayor sea la concentración de valores en la diagonal principal, mejor será el desempeño del descriptor.

```
# Comparación visual en una figura aislada
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)

# Agregar los valores encima de las barras usando round estándar para visualización
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("Análisis de los resultados:")
print("- Píxeles Brutos: Sensibles a variaciones locales de iluminación y ruido.")
print("- LBP: Buena tolerancia para variaciones monótonas de iluminación global.")
print("- HOG: Excelente para contornos y formas estables bajo pequeñas fluctuaciones
geométricas.")
```

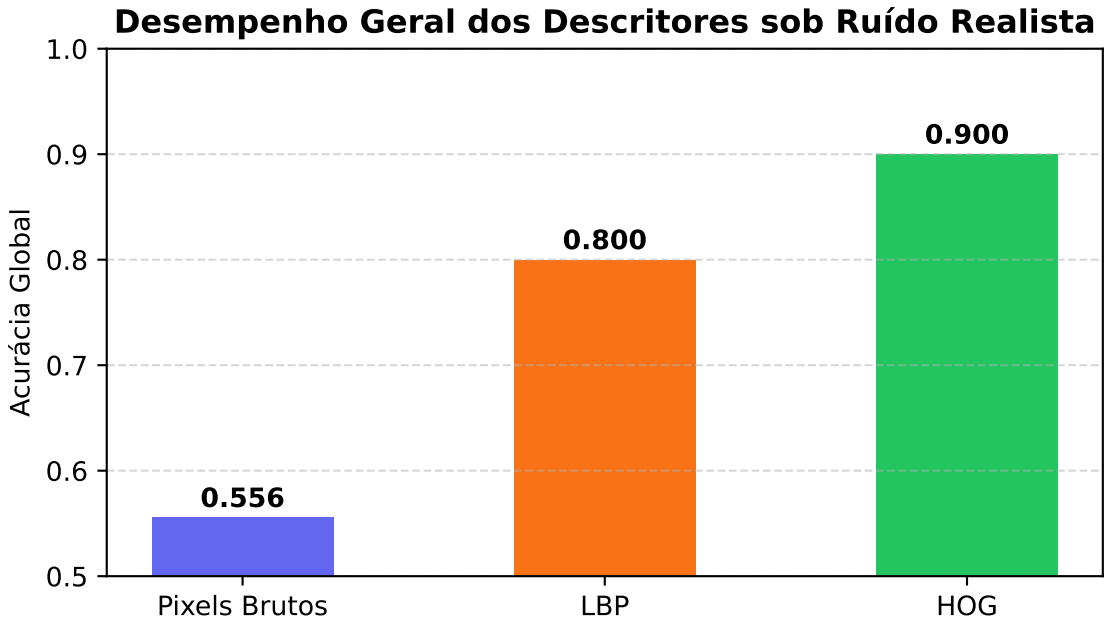


Figura 7.9: Comparación detallada de precisión global en escenario realista. Observe cómo los descriptores extraídos estructuralmente (LBP y HOG) superan el uso de intensidades puras de píxeles brutos.

```
Análisis de los resultados:
- Píxeles Brutos: Sensibles a variaciones locales de iluminación y ruido.
- LBP: Buena tolerancia para variaciones monótonas de iluminación global.
- HOG: Excelente para contornos y formas estables bajo pequeñas fluctuaciones geométricas.
```

7.9 Un Problema Concreto: Simulando Descriptores de Frutas

Retomando el problema de clasificación de frutas presentado al inicio del capítulo, cada imagen puede representarse mediante un vector de características (*feature vector*) obtenido a partir de la extracción de descriptores de color, textura y forma. La Tabla 7.1 presenta algunos descriptores frecuentemente utilizados en aplicaciones de Visión por Computador, incluyendo técnicas introducidas en el Capítulo 3 y en este capítulo.

Tabla 7.1: Conjunto de descriptores cromáticos, texturales y geométricos utilizados para representar imágenes de frutas.

Característica	Descripción
R, G, B	intensidad media de los canales rojo, verde y azul
NC	intensidad media en niveles de gris (*grayscale*)
LBP	descriptor de textura (*Local Binary Pattern*)
HOG	descriptor de forma (*Histogram of Oriented Gradients*)
Área	número de píxeles del objeto
Perímetro	longitud del contorno
Circularidad	medida de cuán circular es el objeto
Relación ancho/alto	proporción entre ancho y alto de la región

En este ejemplo, cada imagen se representa mediante el vector

$$X = (R, G, B, NC, \text{LBP}, \text{HOG}, \text{Área}, \text{Perímetro}, \text{Circularidad}, \text{Razón}).$$

Simplificación adoptada en esta tabla

En la práctica, LBP y HOG no son valores escalares, sino histogramas con decenas o centenas de componentes. En esta sección, cada uno de ellos se representa mediante un único valor solo para simplificar la presentación. En aplicaciones reales, esas posiciones serían sustituidas por las componentes completas de los respectivos histogramas.

En aplicaciones reales, no todos los descriptores contribuyen igualmente a distinguir las clases. Algunos proporcionan información más relevante, mientras que otros pueden ser redundantes o escasamente discriminativos.

Para reproducir este escenario de forma controlada, se utilizará `make_classification()`, de la biblioteca `scikit-learn`. La función genera un conjunto de datos sintético cuyas características pueden interpretarse como descriptores de imágenes, permitiendo definir cuántas de ellas serán informativas para la clasificación.

En este ejemplo, se generan diez características sintéticas, de las cuales solo siete (`n_informative=7`) participan en la separación entre las tres clases. Las restantes simulan atributos poco informativos o redundantes. La Figura 7.10 presenta una representación conceptual de este proceso.

Antes de introducir el algoritmo que se estudiará en detalle en este capítulo, cabe una observación: para identificar, entre las diez características sintéticas, cuáles son más discriminativas — y así seleccionar dos de ellas para la visualización en 2D —, se utiliza un *Random Forest* solo como herramienta auxiliar de diagnóstico. El KNN, foco de este capítulo, se presenta a continuación.

```
# Configuración para reproducción
np.random.seed(42)

# Generación de datos sintéticos con características controladas
X, y = make_classification(
    n_samples=300,
    n_features=10,
    n_informative=7,
    n_redundant=2,
    n_repeated=1,        # Una característica es copia de otra
    n_classes=3,
    n_clusters_per_class=1,
    random_state=42,
)

# Crear figura con dos subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# Subplot 1: Visualización de las clases en 2D (usando dos características informativas)
# Identificar cuáles características son más informativas
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X, y)
importancias = rf.feature_importances_
caracteristicas_informativas = np.argsort(importancias)[-2:]  # Dos más importantes

cores = {0: "#e74c3c", 1: "#f1c40f", 2: "#e67e22"}  # Manzana, Banana, Naranja
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: Importancia de las características
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)

# Colorear barras para destacar características
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)

# Añadir leyenda
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")

# Anotar el número de características informativas
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  Análisis de los datos generados:")
print(f"  • Total de muestras: {X.shape[0]}")
print(f"  • Número de características: {X.shape[1]}")
print(f"  • Características informativas: 7 (columnas 1 a 7 del gráfico)")
print(f"  • Características redundantes: 2 (columnas 8 y 9)")
print(f"  • Características repetidas: 1 (columna 10)")
print(f"  • Distribución de las clases: {np.bincount(y)}")
```

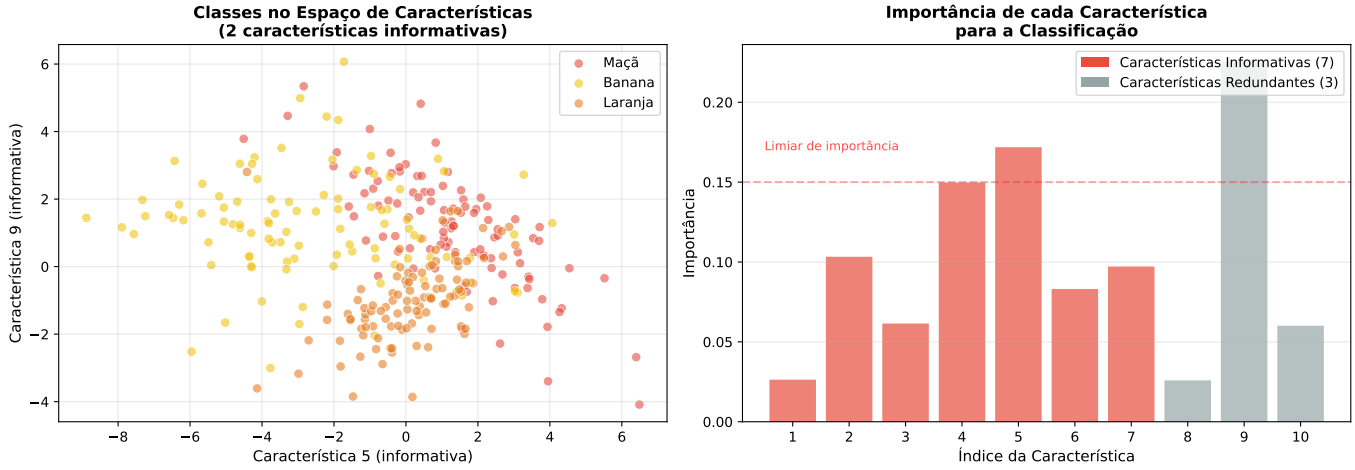


Figura 7.10: Ilustración del proceso de generación de datos sintéticos con *make_classification*. A la izquierda, visualización de las tres clases en un espacio bidimensional formado por dos características informativas. A la derecha, importancia relativa de cada característica para la clasificación, evidenciando que solo 7 de las 10 características son efectivamente discriminativas, mientras que las demás son redundantes (2) o repetidas (1).

```
Análisis de los datos generados:
• Total de muestras: 300
• Número de características: 10
• Características informativas: 7 (columnas 1 a 7 del gráfico)
• Características redundantes: 2 (columnas 8 y 9)
• Características repetidas: 1 (columna 10)
• Distribución de las clases: [101  98 101]
```

7.10 Clasificador k-NN: Cómo Funciona por Dentro

El **k-*Nearest Neighbors* (k-NN)** es uno de los algoritmos de clasificación más simples e intuitivos del Aprendizaje Automático. A diferencia de muchos clasificadores, no construye explícitamente un modelo durante la etapa de entrenamiento. En su lugar, almacena las muestras etiquetadas y, cuando una nueva muestra necesita ser clasificada, busca aquellas que más se le parecen.

El principio del algoritmo se basa en la hipótesis de que las muestras con características similares tienden a pertenecer a la misma clase. Para cuantificar esta proximidad, el k-NN utiliza una medida de distancia entre los vectores de características.

Como ejemplo, considere la Tabla 7.2, que presenta una versión simplificada del problema de clasificación de frutas utilizando solo dos características: intensidad de color y circularidad, ambas normalizadas en el intervalo de 0 a 1.

Tabla 7.2: Ejemplo simplificado de clasificación de frutas utilizando dos características normalizadas.

Muestra	Color	Circularidad	Clase
Fruta 1	0,82	0,88	Manzana
Fruta 2	0,30	0,20	Plátano
Fruta 3	0,88	0,85	Manzana
Fruta ? (prueba)	0,80	0,90	?

Observando solo estas dos características, se percibe que la fruta de prueba está mucho más cerca de las muestras etiquetadas como **Manzana** que de la muestra etiquetada como **Plátano**. En la siguiente sección,

esta noción intuitiva de proximidad se formalizará mediante una métrica de distancia, utilizada por el algoritmo para identificar los vecinos más cercanos y decidir la clase de la nueva muestra.

7.10.1 Métrica de Distancia

La proximidad entre dos muestras se cuantifica normalmente mediante la **distancia euclidiana**, definida por

$$d(x, x_i) = \|x - x_i\|_2 = \sqrt{\sum_{j=1}^{n} (x_j - x_{i,j})^2},$$

donde:

- x es la muestra de prueba;
- x_i es una muestra del conjunto de entrenamiento;
- n es el número de características;
- x_j y $x_{i,j}$ representan la j-ésima característica.

En la implementación de este capítulo, x corresponde a una fila de `X_test` y x_i a una fila de `X_train`. El método `predict()` calcula automáticamente la distancia entre x y todas las muestras de entrenamiento.

En el ejemplo de la Tabla 7.2:

$$d(\text{teste}, \text{Fruta 1}) \approx 0{,}028, \qquad d(\text{teste}, \text{Fruta 2}) \approx 0{,}860, \qquad d(\text{teste}, \text{Fruta 3}) \approx 0{,}094.$$

Como las distancias más pequeñas corresponden a las Frutas 1 y 3, estas muestras se utilizarán en la etapa de decisión.

7.10.2 Regla de Decisión

Tras ordenar las distancias, el algoritmo selecciona los k vecinos más cercanos. Sea $N_k(x)$ ese conjunto. La clase predicha está dada por

$$\hat{y} = \text{moda}\{\, y_i : x_i \in N_k(x) \,\},$$

donde y_i es la etiqueta de la muestra x_i y $\hat{y}$ es la clase asignada a la muestra de prueba.

En el ejemplo, para $k = 3$, los vecinos son Fruta 1 (Manzana), Fruta 3 (Manzana) y Fruta 2 (Banana). Como **Manzana** recibe dos votos, esa es la clase predicha.

Clase `KNeighborsClassifier`

En este capítulo, el algoritmo se implementa con la clase `KNeighborsClassifier`, de la biblioteca `scikit-learn`:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

y_pred = knn.predict(X_test)
```

donde:

- `KNeighborsClassifier(n_neighbors=3)`: define el valor de k;
- `fit(X_train, y_train)`: almacena las muestras de entrenamiento (`X_train`) y sus etiquetas (`y_train`);
- `predict(X_test)`: devuelve las clases predichas para las muestras de `X_test`.

Internamente, `predict()` ejecuta los pasos descritos anteriormente: calcula las distancias, identifica los k vecinos más cercanos y determina la clase por votación mayoritaria.

La Figura 7.11 ilustra este procedimiento en un conjunto bidimensional. La figura destaca los vecinos utilizados en la clasificación, mientras que la consola presenta los pasos del algoritmo: cálculo de las distancias, ordenamiento, selección de los vecinos, votación y predicción de la clase.

```
def knn_passo_a_passo(X, y, x, k=3):
    """Ejecuta los cinco pasos del algoritmo k-NN.
    Parámetros: X (entrenamiento), y (etiquetas), x (prueba) y k (número de vecinos).
    """
    # 1. Distancias
    dist = [(np.linalg.norm(x-xi), yi, i) for i, (xi, yi) in enumerate(zip(X, y))]
    print(f"1. Distancias calculadas: {len(dist)}")

    # 2. Ordenación
    dist.sort(key=lambda t: t[0])
    print("2. Distancias ordenadas")

    # 3. Selección
    vizinhos = dist[:k]
    print(f"3. {k} vecinos más cercanos:")
    for d, c, _ in vizinhos:
        print(f"   {d:.4f} → {c}")

    # 4. Votación
    votos = {}
    for _, c, _ in vizinhos:
        votos[c] = votos.get(c, 0) + 1
    print("4. Votos:", votos)

    # 5. Decisión
    classe = max(votos, key=votos.get)
    print("5. Clase predicha:", classe)

    return classe, vizinhos


# Datos de ejemplo
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)

# Visualización
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. Distancias calculadas: 30
2. Distancias ordenadas
3. 3 vecinos más cercanos:
   1.0850 → Classe A
   1.6379 → Classe B
   1.6382 → Classe A
4. Votos: {np.str_('Classe A'): 2, np.str_('Classe B'): 1}
5. Clase predicha: Classe A
```

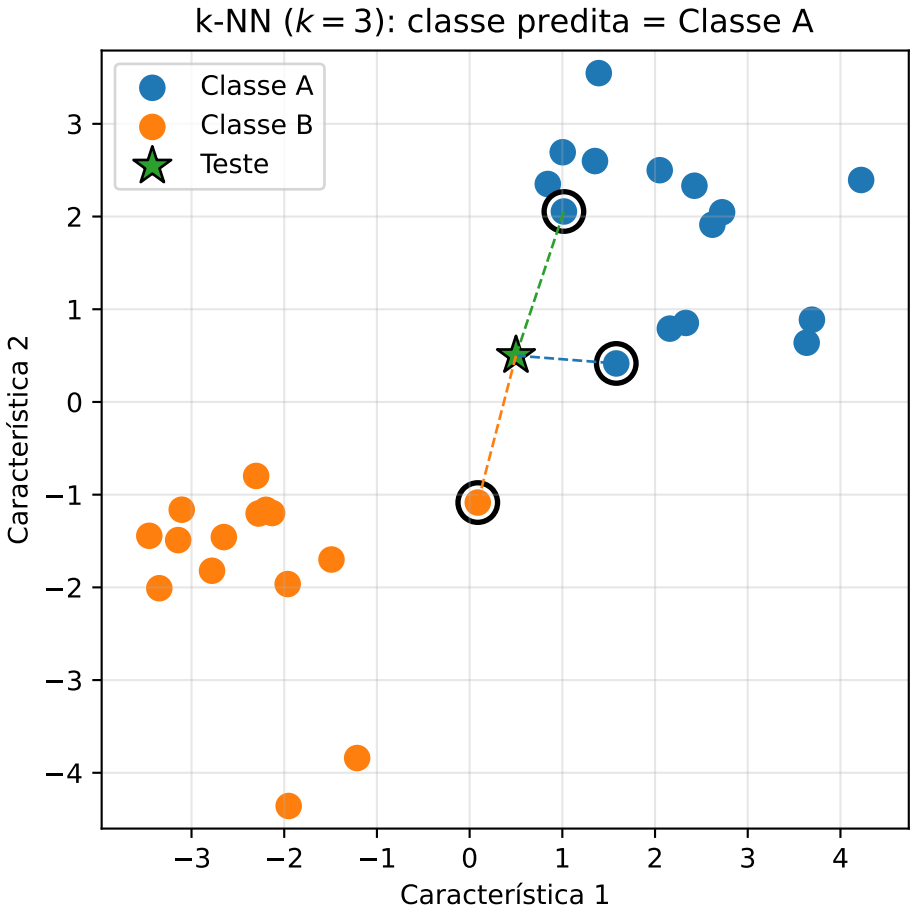


Figura 7.11: Clasificación de una nueva muestra por el algoritmo k-NN. El punto de prueba (estrella) se clasifica a partir de los tres vecinos más cercanos, resaltados por círculos.

7.10.3 El Papel del Parámetro k

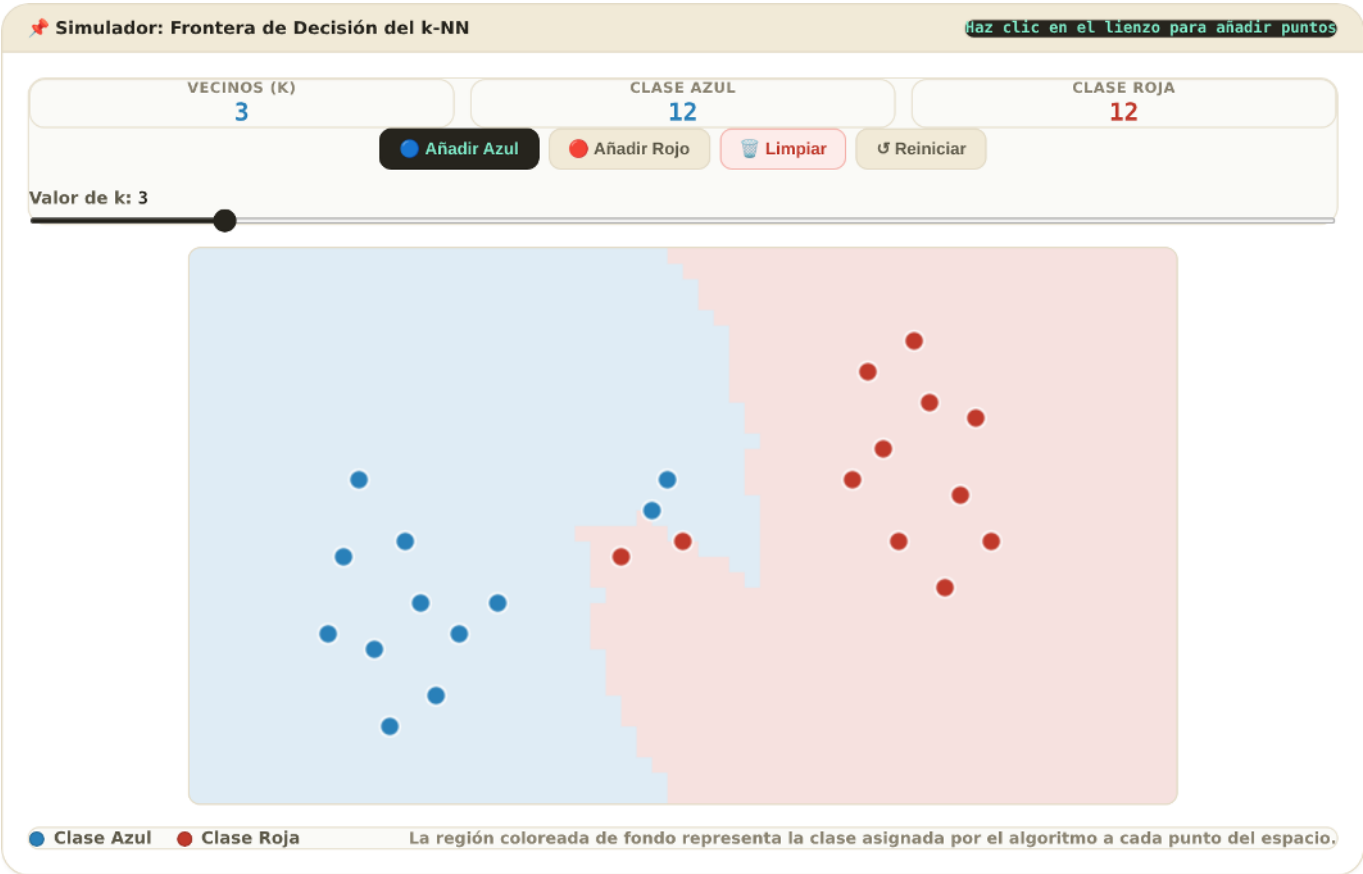
El parámetro k determina cuántos vecinos participan en la decisión de clasificación.

- **Valores pequeños de** k (por ejemplo, $k = 1$) hacen que el clasificador sea más sensible a ruidos y variaciones locales, produciendo fronteras de decisión más irregulares y favoreciendo el *sobreajuste*.
- **Valores mayores de** k producen fronteras de decisión más suaves, pero pueden reducir la sensibilidad a estructuras locales, favoreciendo el *subajuste*.

En problemas con dos clases, es común utilizar valores impares de k para reducir la ocurrencia de empates.

Otro aspecto importante es la **maldición de la dimensionalidad** (*curse of dimensionality*). A medida que el número de características aumenta, las distancias entre las muestras tienden a volverse más similares, dificultando la identificación de vecinos realmente representativos.

> **Resumen**
>
> El algoritmo k-NN puede resumirse en tres etapas:
> 1. extraer el vector de características de la nueva muestra;
> 2. identificar los k vecinos más cercanos;
> 3. clasificar la muestra por la clase más frecuente entre esos vecinos.
>
> El simulador de la Figura 7.12 permite explorar visualmente el efecto del parámetro k sobre la frontera de decisión.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap07/sim-07-knn.html

Figura 7.12: Simulador interactivo de la frontera de decisión del k-NN: añada puntos de entrenamiento y ajuste el valor de k para observar el efecto sobre la región de decisión.

7.11 Proyecto Práctico 1: Clasificación de Dígitos Manuscritos con k-NN

Las secciones anteriores presentaron el algoritmo k-NN mediante un ejemplo simplificado de clasificación de frutas, utilizando únicamente dos características. A continuación, el mismo algoritmo se aplica a un conjunto de datos de imágenes, en el cual cada muestra está representada por un vector de mayor dimensión.

Como estudio de caso, se utiliza la base pública `load_digits`, disponible en la biblioteca `scikit-learn`. Este conjunto de datos contiene 1797 imágenes de dígitos manuscritos de las clases de 0 a 9, cada una con una resolución de 8×8 píxeles en niveles de gris. Cada imagen está representada por un vector con 64 características, correspondientes a las intensidades de los píxeles, y cada vector posee una etiqueta que indica el dígito correspondiente.

La base `load_digits` está disponible en la biblioteca `scikit-learn` y se utiliza en este capítulo para ilustrar la aplicación del algoritmo k-NN. Además de estar disponible directamente en `scikit-learn`, evita etapas adicionales de obtención y preparación de los datos, permitiendo concentrar la atención en la implementación y la evaluación del clasificador.

La Figura 7.13 presenta una muestra de las imágenes de la base de datos.

```
digits = load_digits()
print(f"Total de muestras: {digits.data.shape[0]}, dimensión del vector:
{digits.data.shape[1]}")
print(f"Clases: {[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 de muestras: 1797, dimensión del vector: 64
Clases: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

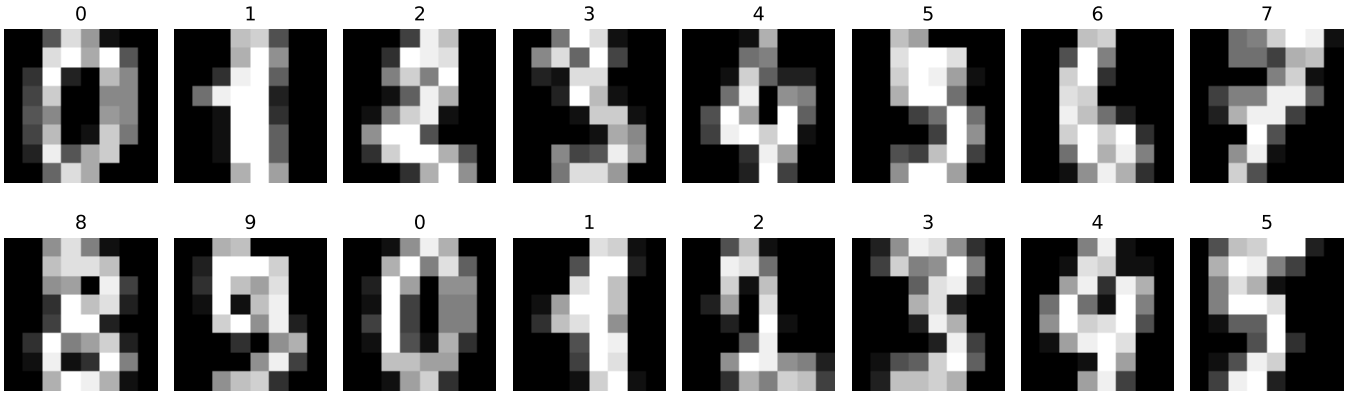


Figura 7.13: Muestra de dígitos manuscritos de la base *load_digits*, utilizada como caso de estudio de clasificación.

7.11.1 Clasificación con Vectores de Intensidad

En este primer experimento, cada imagen de dimensión 8×8 se representa directamente mediante las intensidades de sus 64 píxeles, sin la extracción de descriptores adicionales. Así, cada muestra corresponde a un vector de 64 características, utilizado como entrada del clasificador k-NN.

A continuación, el conjunto de datos se divide en subconjuntos de entrenamiento y prueba, preservando la proporción de las diez clases mediante el parámetro `stratify=y`. El clasificador se entrena con $k = 3$ y se evalúa sobre el conjunto de prueba utilizando la exactitud y la matriz de confusión presentada en la Figura 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"Precisión (vectores de intensidad, 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()
```

```
Precisión (vectores de intensidad, k=3): 0.9870
```

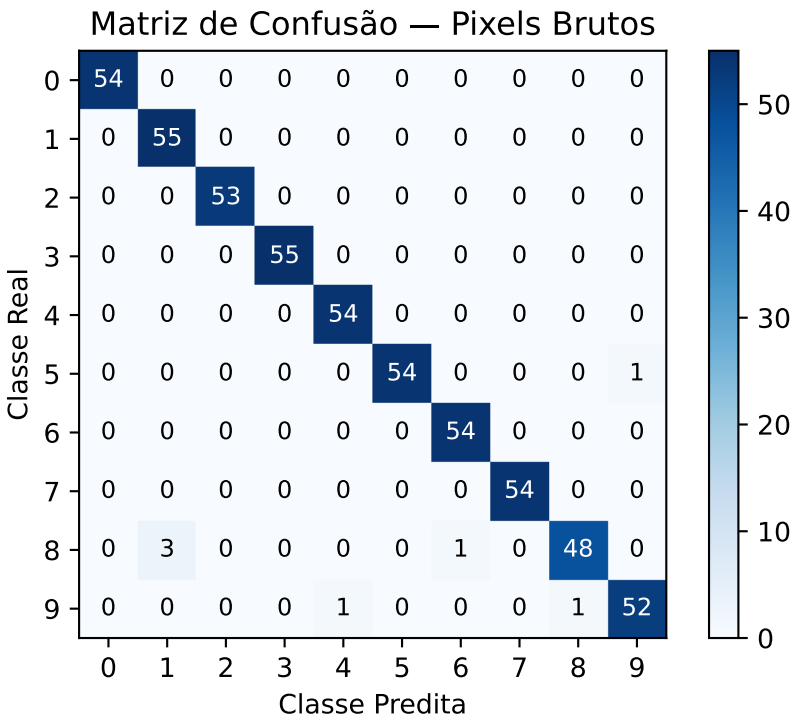


Figura 7.14: Matriz de confusión del clasificador k-NN entrenado con vectores de intensidad brutos (píxeles).

7.11.2 Clasificación con Descriptores HOG

En el experimento anterior, cada imagen fue representada directamente por las intensidades de sus píxeles. En esta sección, esa representación se sustituye por descriptores HOG (*Histogram of Oriented Gradients*), que codifican información sobre la distribución de las orientaciones de los gradientes de la imagen.

Se mantienen la misma partición de los datos, el mismo clasificador k-NN y el mismo protocolo de evaluación, cambiando únicamente la representación de las imágenes. La Figura 7.15 compara los resultados obtenidos con vectores de intensidad y con descriptores HOG.

```
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"Dimensión del vector HOG: {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"Precisión (descriptor HOG, 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)  # Aumenta el límite superior para dar espacio
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")  # Aumenta el desplazamiento vertical
plt.tight_layout()
```

```
Dimensión del vector HOG: 32
Precisión (descriptor HOG, k=3): 0.7593
```

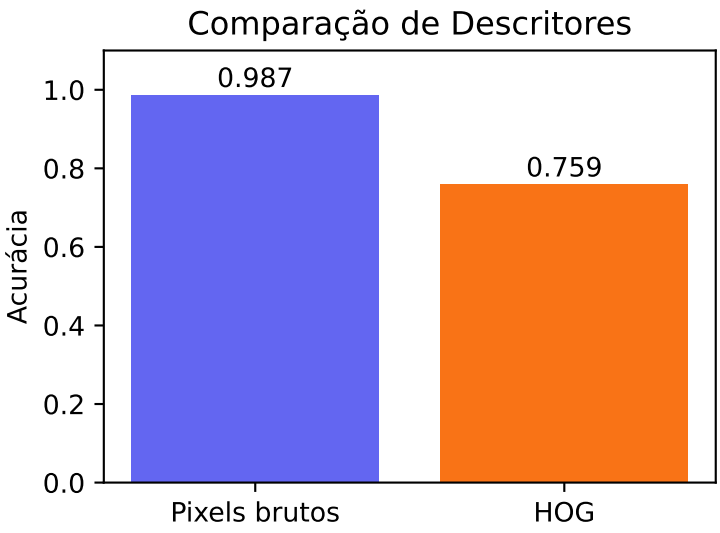


Figura 7.15: Comparación de precisión entre descriptores de píxeles brutos y HOG para el clasificador k-NN (k=3) en la base de dígitos.

¿Por qué sucede esto?

En Figura 7.15, el clasificador entrenado con vectores de **intensidad de los píxeles** alcanza una mayor exactitud (0.987) que aquel basado en descriptores **HOG** (0.759). Este resultado está relacionado con las características de la base `load_digits`.
Las imágenes poseen una resolución de solo 8×8 píxeles, se encuentran aproximadamente centradas y presentan poca variación de iluminación, escala y orientación. En este escenario, las intensidades de los píxeles preservan prácticamente toda la información necesaria para distinguir las clases. En contraste, el HOG resume la imagen en histogramas de orientaciones de los gradientes, reduciendo parte del detalle espacial disponible en los píxeles originales.
Esta reducción de información puede dificultar la separación de dígitos visualmente similares, como 3 y 8 o 4 y 9, especialmente cuando la resolución de la imagen es baja.
En problemas con imágenes de mayor resolución o sujetas a variaciones de iluminación, posición, escala o pequeñas deformaciones, descriptores como el HOG tienden a representar mejor la estructura local de la imagen que los valores individuales de los píxeles. Así, este experimento ilustra un principio importante del Aprendizaje Automático: **la representación de los datos debe elegirse de acuerdo con las características del problema y no por la complejidad del descriptor.**

7.12 Evaluación de Clasificadores

En las secciones anteriores, la calidad del clasificador fue analizada mediante la exactitud y la matriz de confusión. En esta sección, estas herramientas se complementan con métricas utilizadas en la evaluación de modelos y con un procedimiento para seleccionar el valor del parámetro k.

La exactitud corresponde a la proporción de muestras clasificadas correctamente. Aunque es una medida simple y ampliamente utilizada, puede ser insuficiente cuando las clases presentan distribuciones muy desbalanceadas.

A partir de la matriz de confusión — introducida en el **Capítulo 1** y utilizada a lo largo de este capítulo — se pueden calcular métricas por clase, como precisión y exhaustividad:

$$\text{Precisión} = \frac{VP}{VP + FP}, \qquad \text{Exhaustividad} = \frac{VP}{VP + FN},$$

donde VP, FP y FN representan, respectivamente, el número de verdaderos positivos, falsos positivos y falsos negativos de la clase analizada. La precisión cuantifica la proporción de predicciones positivas correctas, mientras que la exhaustividad mide la capacidad del clasificador de identificar los ejemplos pertenecientes a la clase.

7.12.1 Elección de k por Validación Cruzada

En los experimentos anteriores, se adoptó $k = 3$ para ilustrar el funcionamiento del algoritmo. Sin embargo, este parámetro influye directamente en el desempeño del clasificador y, en la práctica, debe seleccionarse a partir de los datos.

Un enfoque ampliamente utilizado es la **validación cruzada** (*cross-validation*), en la cual el conjunto de entrenamiento se divide en particiones sucesivas para estimar el desempeño del modelo en datos no utilizados durante el entrenamiento.

El siguiente código calcula la precisión media obtenida por validación cruzada de cinco particiones (*5-fold cross-validation*) para diferentes valores de k. La Figura 7.16 presenta los resultados, permitiendo identificar la región en la que el clasificador alcanza un mejor desempeño.

```
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"Mejor valor de k encontrado: {melhor_k} (precisión
media={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()
```

```
Mejor valor de k encontrado: 2 (precisión media=0.9672)
```

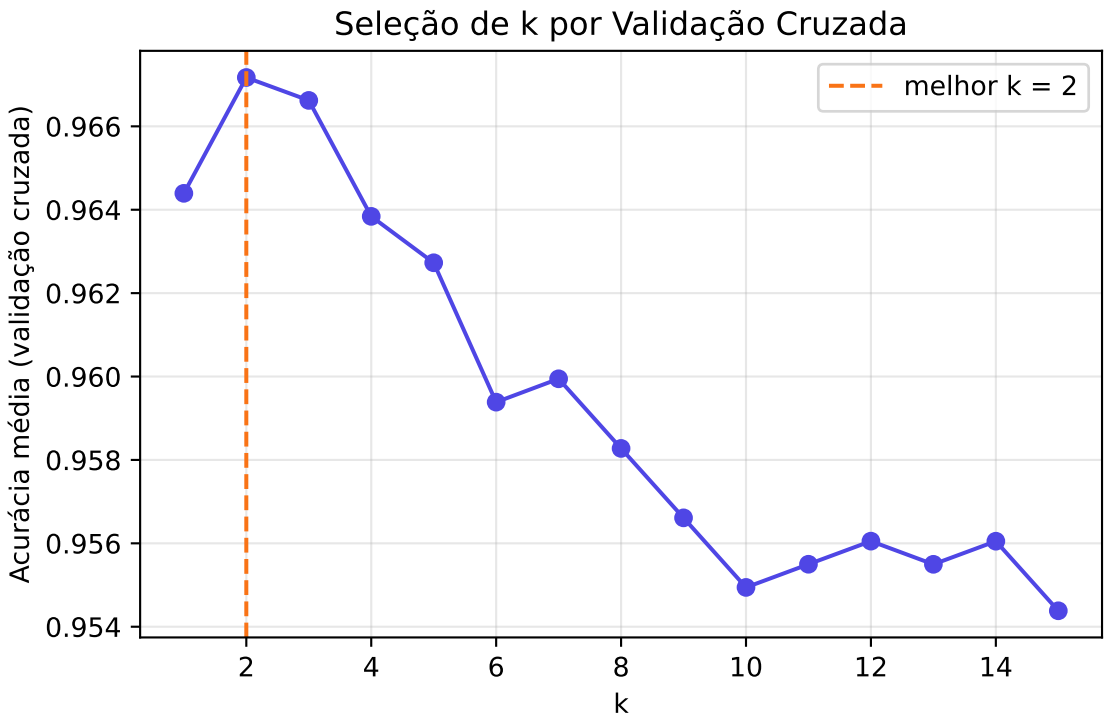


Figura 7.16: Precisión media por validación cruzada (5 particiones) en función del parámetro k, para la base de dígitos con vectores de intensidad brutos.

7.12.2 El Compromiso entre Sesgo y Varianza: Diagnóstico de *Overfitting* y *Underfitting*

El hiperparámetro k influye en la complejidad de la frontera de decisión del clasificador k-NN y, en consecuencia, en su capacidad de generalización. En términos generales, valores pequeños de k hacen que el modelo sea más sensible a las muestras de entrenamiento, mientras que valores mayores producen fronteras de decisión más suaves.

Estos comportamientos están asociados con el compromiso entre **sesgo** (*bias*) y **varianza** (*variance*). Valores muy pequeños de k tienden a aumentar el riesgo de **sobreajuste** (*overfitting*), especialmente en conjuntos de

datos ruidosos, mientras que valores muy grandes pueden llevar al **subajuste** (*underfitting*), reduciendo la capacidad del modelo para capturar estructuras locales de los datos.

Mientras que la Figura 7.16 presentó solo la precisión media obtenida mediante validación cruzada, la Figura 7.17 compara las precisiones de entrenamiento y de prueba para diferentes valores de k. Las regiones resaltadas en el gráfico representan el comportamiento esperado del algoritmo: mayor riesgo de sobreajuste para valores pequeños de k, una región intermedia que frecuentemente produce un buen equilibrio entre sesgo y varianza, y un mayor riesgo de subajuste para valores elevados de k.

Sin embargo, estas regiones deben interpretarse solo como una referencia conceptual. El comportamiento observado depende de las características del conjunto de datos. En la base `load_digits`, por ejemplo, las imágenes presentan poca variabilidad y una buena separación entre las clases, de modo que valores pequeños de k pueden presentar un rendimiento similar —o incluso superior— a los demás, sin evidenciar un sobreajuste significativo.

```
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("Interpretación")
print("- Valores pequeños de k: mayor riesgo de overfitting.")
print("- Valores intermedios: mejor compromiso entre sesgo y varianza.")
print("- Valores grandes de k: mayor riesgo de underfitting.")
print("\nLas regiones coloreadas representan tendencias generales;")
print("el comportamiento observado depende del conjunto de datos.")
print(f"\nMayor precisión en la prueba: {maior_acc:.3f}")
print(f"Valores de k que alcanzaron esa precisión: {melhores_k}")
```

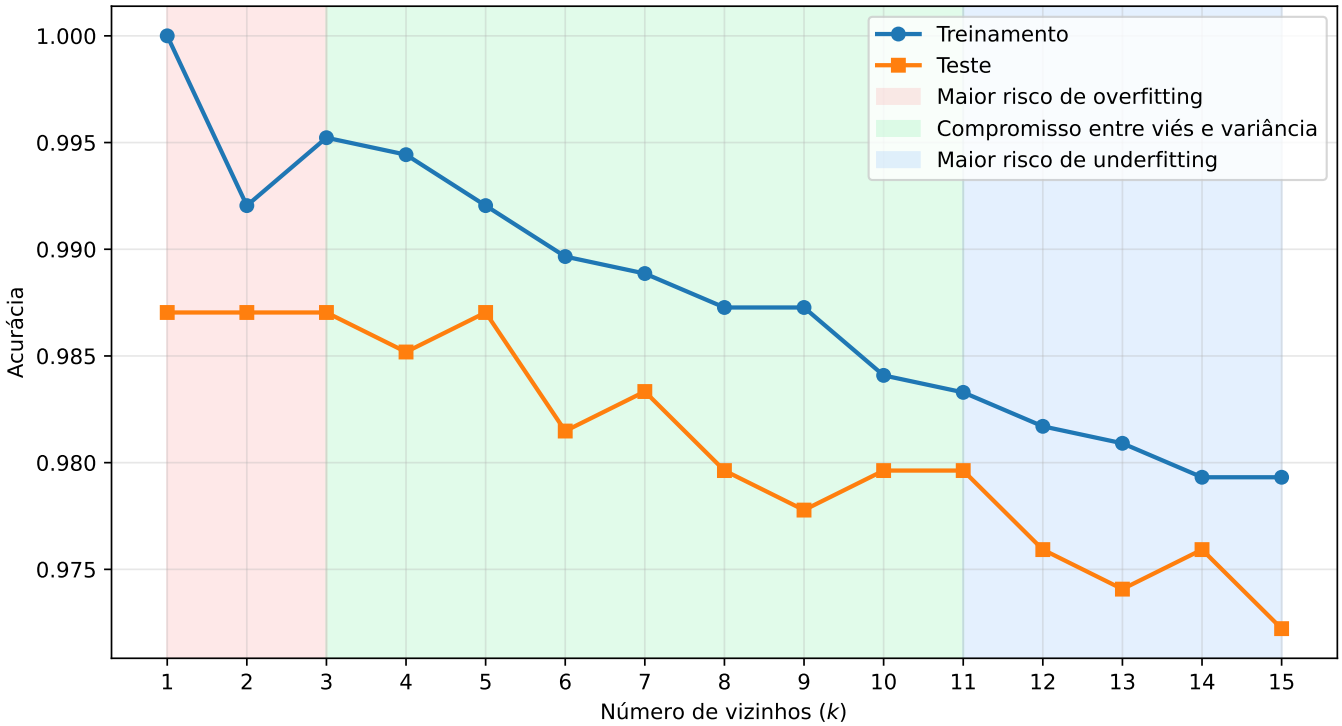


Figura 7.17: Precisión en los conjuntos de entrenamiento y prueba para diferentes valores de k. Las regiones coloreadas representan, de forma conceptual, tendencias de comportamiento del clasificador: mayor riesgo de sobreajuste (rojo), compromiso entre sesgo y varianza (verde) y mayor riesgo de subajuste (azul).

```
Interpretación
- Valores pequeños de k: mayor riesgo de overfitting.
- Valores intermedios: mejor compromiso entre sesgo y varianza.
- Valores grandes de k: mayor riesgo de underfitting.

Las regiones coloreadas representan tendencias generales;
el comportamiento observado depende del conjunto de datos.

Mayor precisión en la prueba: 0.987
Valores de k que alcanzaron esa precisión: [1, 2, 3, 5]
```

7.13 Proyecto Práctico 2: Comparación de Descriptores para Clasificación de Texturas

En el **Capítulo 6**, la varianza local se utilizó como descriptor de textura para **detectar** anomalías en superficies industriales, distinguiendo muestras **conformes** y **defectuosas**. En este proyecto, el problema se reformula como una tarea de **clasificación multiclase**, en la cual diferentes representaciones de la imagen se utilizan como entrada para un clasificador.

Se considerarán tres tipos de descriptores: las **intensidades de los píxeles**, el **Local Binary Patterns (LBP)** y el **Histogram of Oriented Gradients (HOG)**. Para cada representación, se extraerá un vector de características que servirá de entrada para el algoritmo de los k vecinos más cercanos (k-NN). Al final, se compararán las precisiones obtenidas por cada descriptor en las condiciones definidas para este experimento.

La evaluación se realizará mediante **validación cruzada estratificada en cinco particiones (*5-fold stratified cross-validation*)**. En este procedimiento, el conjunto de datos se divide en cinco subconjuntos preservando la proporción entre las clases. En cada iteración, una partición se utiliza para prueba y las cuatro restantes para entrenamiento, repitiéndose el proceso hasta que todas las particiones hayan sido utilizadas como conjunto de prueba. Al término de las cinco ejecuciones, se calculan la precisión media y la desviación

estándar para cada descriptor.

El conjunto de datos está compuesto por tres clases de texturas sintéticas: **granular**, obtenida a partir de ruido gaussiano suavizado; **listada**, formada por patrones sinusoidales periódicos; y **manchada**, compuesta por regiones circulares superpuestas. Para introducir variabilidad entre las muestras, todas las imágenes reciben una perturbación por ruido gaussiano de baja intensidad. La Figura 7.18 presenta ejemplos de las tres clases utilizadas en el experimento.

```
rng = np.random.default_rng(42)

def gerar_textura(classe, tamanho=64, ruido=0.10):
    """Genera una textura sintética de 64x64 perteneciente a una de las tres clases."""
    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))
```

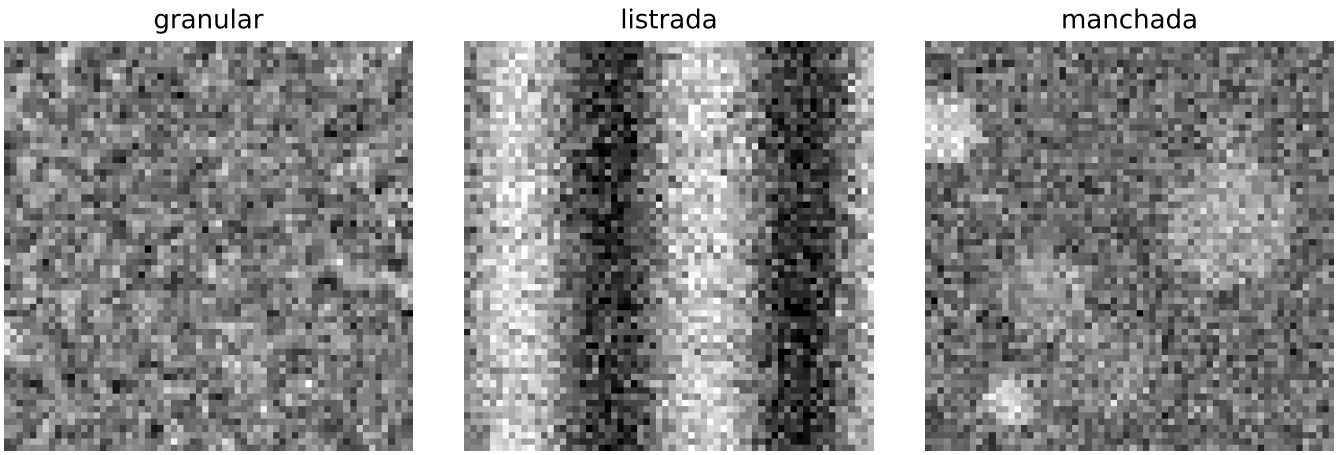


Figura 7.18: Muestras sintéticas de las tres clases de textura utilizadas en el experimento de clasificación, generadas con ruido gaussiano y variabilidad intra-clase.

7.13.1 *Pipeline* de Extracción de Características y Evaluación Comparativa

Para comparar el rendimiento de diferentes formas de representación de las imágenes, se generó un conjunto de datos balanceado que contiene 60 muestras para cada clase. A partir de este conjunto, se extrajeron tres tipos de vectores de características, cada uno representando aspectos distintos de la información visual:

1. **Píxeles crudos**: vector obtenido mediante el aplanamiento (*flattening*) de la matriz de intensidades de la imagen, resultando en un vector de $64 \times 64 = 4096$ atributos;
2. **Histograma LBP uniforme**: histograma normalizado de las frecuencias de los patrones locales producidos por el operador LBP uniforme, compuesto por 10 atributos;
3. **Descriptor HOG**: vector formado por histogramas de gradientes orientados, que representan la distribución espacial de las orientaciones de los bordes, totalizando 128 atributos.

Como estos descriptores poseen escalas y dimensionalidades distintas, los vectores de características se estandarizan utilizando el `StandardScaler`, de modo que cada atributo presente media cero y desviación estándar unitaria. Esta etapa evita que atributos con mayor amplitud influyan desproporcionadamente en el cálculo de las distancias euclidianas empleado por el clasificador.

La evaluación se realiza utilizando el algoritmo k-NN con $k = 5$, bajo el mismo protocolo de validación cruzada estratificada en cinco particiones descrito en la sección anterior. La precisión media obtenida a lo largo de las cinco ejecuciones, ver Figura 7.19, proporciona una estimación más estable del rendimiento del clasificador, reduciendo la dependencia de una única división entre entrenamiento y prueba.

```
rng = np.random.default_rng(42)

# 1. Generación de la base de datos
X_bruto, X_lbp, X_hog, y_textura = [], [], [], []

for classe in classes_textura:
    for _ in range(60):
        img = gerar_textura(classe, ruido=0.10)

        # Extracción 1: Píxeles brutos
        X_bruto.append(img.ravel())

        # Extracción 2: Histograma LBP uniforme
        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)
```

```
        # Extracción 3: Descriptor HOG
        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. Evaluación estadística mediante validación cruzada de 5 pliegues
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} -> Precisión Media: {scores.mean():.4f} (± {scores.std():.4f})")

# 3. Trazado del gráfico comparativo formal
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   -> Precisión Media: 0.6889 (± 0.0478)
LBP (Textura)   -> Precisión Media: 0.7611 (± 0.0648)
HOG (Forma)     -> Precisión Media: 0.4889 (± 0.0624)
```

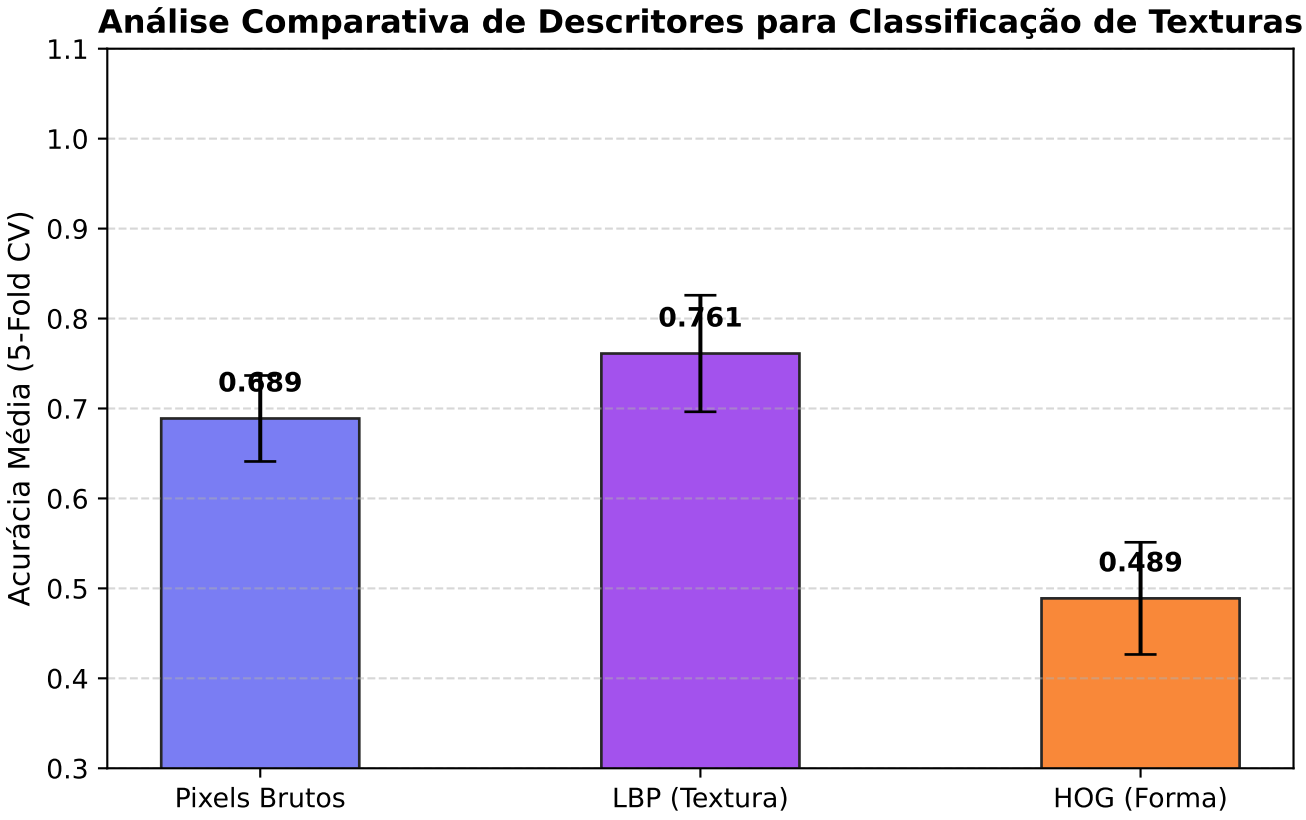


Figura 7.19: Precisión media obtenida mediante validación cruzada (5-fold) para los descriptores de Píxeles Brutos, LBP y HOG aplicados a la base de texturas sintéticas.

¿Por qué funciona? — LBP como representación de texturas

El descriptor LBP representa la textura de una imagen mediante un histograma normalizado que contabiliza la frecuencia de los patrones locales de intensidad. En lugar de almacenar directamente los valores de los píxeles o sus posiciones, esta representación resume la distribución de las microestructuras presentes en la imagen, produciendo un vector de características compacto.

En este experimento se compararon tres tipos de descriptores: píxeles brutos, LBP y HOG. Los vectores formados por los **píxeles brutos** preservan todas las intensidades de la imagen, pero también incorporan variaciones derivadas del ruido y de pequeños desplazamientos espaciales, lo que puede dificultar la comparación entre muestras mediante la distancia euclidiana.

El descriptor **HOG** representa la distribución de las orientaciones de los gradientes, siendo adecuado para describir formas y contornos. Dado que las imágenes utilizadas en este proyecto difieren principalmente por sus propiedades de textura, y no por la presencia de contornos bien definidos, esta representación tiende a capturar menos información discriminativa que el LBP.

En cambio, el **LBP** fue desarrollado específicamente para caracterizar patrones locales de textura. Su histograma describe la frecuencia de las microestructuras presentes en la imagen, independientemente de su posición exacta, haciendo que la representación sea menos sensible a pequeñas variaciones espaciales y a cambios monótonos de iluminación.

Aunque los histogramas de las diferentes clases presentan distribuciones distintas, el ruido gaussiano introducido en la generación de las imágenes aumenta la variabilidad entre muestras de la misma clase y puede producir regiones de solapamiento en el espacio de características. Como consecuencia, algunas texturas pueden ser confundidas por el clasificador. Aun así, cuando las características relevantes para distinguir las clases están asociadas a los patrones locales de textura, se espera que descriptores diseñados para ese fin, como el LBP, produzcan representaciones más informativas que aquellas basadas únicamente en las intensidades de los píxeles o en las orientaciones de los gradientes.

7.13.2 Diagnóstico Fino del Clasificador: Precisión, Exhaustividad y Puntuación F1

La exactitud resume el desempeño del clasificador en un único valor, pero no indica cómo se distribuye ese desempeño entre las diferentes clases. Para un análisis más detallado, se utilizan métricas calculadas individualmente para cada clase.

La **precisión** (*precision*) mide la proporción de muestras clasificadas como pertenecientes a una clase que realmente pertenecen a ella. La **exhaustividad** (*recall*) mide la proporción de muestras de la clase que fueron correctamente identificadas por el clasificador. La **puntuación F1** corresponde a la media armónica entre precisión y exhaustividad, proporcionando un indicador que equilibra ambas medidas.

El informe también indica el **soporte** (*support*), es decir, el número de muestras de cada clase presentes en el conjunto de prueba. Esta información es importante para contextualizar las métricas, pues los resultados obtenidos sobre pocas muestras tienden a presentar una mayor variabilidad.

La Figura 7.20 presenta estas métricas para las tres clases de textura. En conjunto, permiten identificar diferencias de desempeño que no son evidentes solo a partir de la exactitud. Por ejemplo, una clase puede presentar alta precisión y menor exhaustividad, lo que indica que el clasificador comete pocos falsos positivos, pero deja de identificar parte de las muestras que realmente pertenecen a esa clase. Este tipo de análisis ayuda a comprender las limitaciones del modelo y a identificar posibles estrategias para su mejora.

```
X_lbp_data = np.array(X_lbp)
y_textura_data = np.array(y_textura)

# Realizar una división de entrenamiento/prueba para el informe detallado
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
)

# Escalar los datos
scaler = StandardScaler()
Xt_treino_scaled = scaler.fit_transform(Xt_treino)
Xt_teste_scaled = scaler.transform(Xt_teste)

# Entrenar el clasificador k-NN
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("=== INFORME DE CLASIFICACIÓN DETALLADO ===")
print(f"{'Classe':<12} {'Precisão':>10} {'Revocação':>12} {'F1-score':>10} {'Suporte':>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("Interpretación de las métricas:")
print("- Precisión: entre las muestras clasificadas como pertenecientes a la clase,",
      "¿cuántas eran correctas?")
print("- Revocación: entre las muestras que realmente pertenecen a la clase, ¿cuántas",
      "fueron identificadas?")
print("- Puntuación F1: media armónica entre precisión y revocación.")
print("- Soporte: número de muestras reales de cada clase presentes en el conjunto de
prueba.")
print("\nEl soporte no mide el rendimiento; solo informa cuántos ejemplos de cada",
      "clase fueron \nutilizados en la evaluación.")
```

```
=== INFORME DE CLASIFICACIÓN DETALLADO ===
Classe          Precisão    Revocação   F1-score    Suporte
granular            0.67        0.78       0.72         18
listrada            0.55        0.61       0.58         18
manchada            1.00        0.72       0.84         18
```

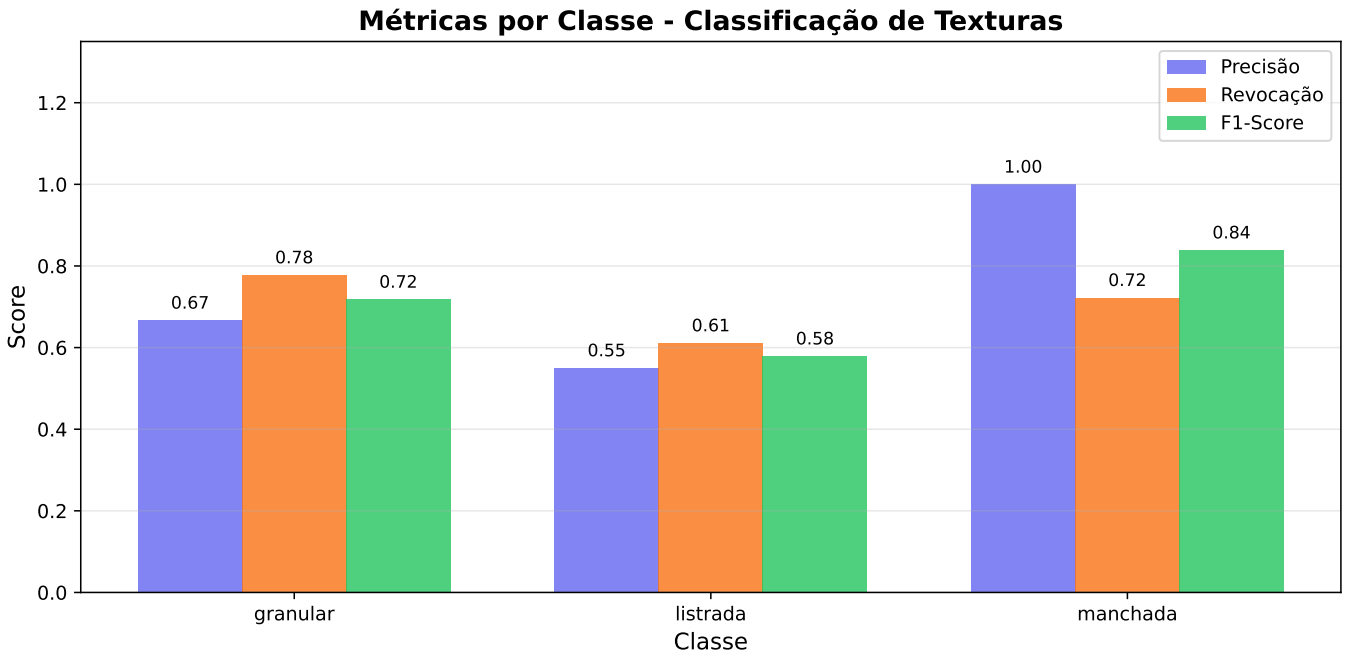


Figura 7.20: Métricas de evaluación detalladas para el clasificador k-NN con descriptores LBP

```
Interpretación de las métricas:
- Precisión: entre las muestras clasificadas como pertenecientes a la clase, ¿cuántas eran
correctas?
- Revocación: entre las muestras que realmente pertenecen a la clase, ¿cuántas fueron
identificadas?
- Puntuación F1: media armónica entre precisión y revocación.
- Soporte: número de muestras reales de cada clase presentes en el conjunto de prueba.

El soporte no mide el rendimiento; solo informa cuántos ejemplos de cada clase fueron
utilizados en la evaluación.
```

7.14 Limitaciones de los Descriptores Artesanales

Los experimentos de este capítulo muestran que los descriptores clásicos pueden ser bastante eficaces en tareas de clasificación, pero también presentan limitaciones importantes:

- **Especificidad:** cada descriptor fue desarrollado para representar un determinado tipo de información, como color, textura o forma. Así, un descriptor adecuado para una tarea puede no ser el más apropiado para otra.
- **Dependencia de hiperparámetros:** el rendimiento de descriptores como LBP y HOG depende de la elección de parámetros, como radio de vecindad, número de puntos muestreados, tamaño de la celda y número de orientaciones, que deben ajustarse según la aplicación.
- **Representación limitada:** los descriptores de color, textura y gradiente capturan propiedades de bajo nivel de la imagen, pero no representan directamente conceptos semánticos más complejos, como objetos o escenas.
- **Maldición de la dimensionalidad:** los descriptores muy extensos pueden reducir la eficacia de los clasificadores basados en distancia, como el k-NN.

Estas limitaciones motivan la evolución de las técnicas estudiadas en los próximos capítulos. El **Capítulo 8** presenta métodos clásicos para la detección y correspondencia de características en imágenes, mientras que el **Capítulo 9** introduce las **Redes Neuronales Convolucionales**, capaces de aprender automáticamente representaciones adecuadas para cada tarea a partir de los datos.

7.15 Resumen

En este capítulo se presentaron los fundamentos del reconocimiento de patrones aplicado a imágenes. Los principales conceptos estudiados fueron:

- ***Pipeline* de reconocimiento de patrones:** adquisición, preprocesamiento, extracción de descriptores, clasificación y evaluación.
- **Descriptores clásicos:** descriptores de color, LBP para textura y HOG para forma, utilizados para representar diferentes características de las imágenes.
- **Normalización de características:** estandarización (*Z-score*) para evitar que atributos de mayor magnitud dominen el cálculo de las distancias.
- **Clasificador k-NN:** clasificación basada en los k vecinos más cercanos en el espacio de características.
- **Elección del parámetro k:** influencia del valor de k sobre el rendimiento del clasificador y uso de la validación cruzada para su selección.
- **Evaluación de clasificadores:** exactitud, matriz de confusión, precisión, recuperación y puntuación F1 como métricas complementarias de rendimiento.
- **Limitaciones de los descriptores artesanales:** especificidad, dependencia de hiperparámetros y dificultad para representar información de alto nivel.

Los conceptos se ilustraron mediante experimentos con la base pública `load_digits`, texturas sintéticas generadas con fines didácticos y datos simulados de descriptores de frutas.

7.16 🤖 Uso del Gemini Notebook como Tutor

En esta edición, el **Gemini Notebook** se presenta como una herramienta de apoyo al estudio. El sistema utiliza exclusivamente los documentos proporcionados por el autor como fuente de conocimiento, permitiendo explorar los conceptos del capítulo mediante preguntas, resúmenes y explicaciones relacionadas con el material estudiado.

🎓 Estudia con el Tutor Inteligente

🚀 ACCEDER AL GEMINI NOTEBOOK: CAPÍTULO 07

🌐 **Idioma y Lenguaje de Programación**

El proyecto de este capítulo en el Gemini Notebook fue construido únicamente con el texto en **portugués** y los ejemplos de código en **Python**. Si estás estudiando con la edición en inglés o francés, o siguiendo la ruta en C++, las respuestas del tutor pueden no corresponder exactamente con la versión que estás leyendo.

⚠️ **Aviso sobre Contenido Generado por IA**

Las respuestas proporcionadas por el Gemini Notebook pueden contener imprecisiones u omisiones. Siempre que sea necesario, confirma la información utilizando el material de este capítulo y otras fuentes académicas confiables. La ejecución de los ejemplos prácticos presentados a lo largo del texto sigue siendo la mejor forma de consolidar los conceptos estudiados.

7.17 Lista de Ejercicios

Los ejercicios siguientes exploran y extienden los conceptos presentados en este capítulo mediante adaptaciones de los algoritmos implementados, análisis experimentales y comparaciones entre diferentes enfoques.

1. **(10%)** Implemente un descriptor de color (histograma RGB o HSV, con al menos 16 *bins* por canal) para las tres clases de frutas simuladas en la Figura 7.1. Entrene un clasificador k-NN con este descriptor,

compare su exactitud con la obtenida por los descriptores LBP y HOG (Figura 7.9) y discuta en qué situaciones la información de color es más discriminativa.

2. **(15%)** Investigue el efecto de la normalización de características (*Z-score*) sobre el desempeño del k-NN en un espacio de atributos heterogéneo, combinando descriptores de color, LBP y HOG en un único vector. Compare los resultados obtenidos con y sin normalización para al menos tres valores de k.

3. **(15%)** Reproduzca el análisis de *overfitting* y *underfitting* de la Figura 7.17 variando el tamaño del conjunto de entrenamiento (por ejemplo, 20%, 50% y 80% de la base `load_digits`). Discuta cómo la cantidad de ejemplos influye en la elección del valor de k.

4. **(15%)** Extienda el Proyecto Práctico 2 añadiendo una cuarta clase sintética de textura. Evalúe precisión, exhaustividad y F1-score para cada clase, siguiendo el patrón de la Figura 7.20, y analice el impacto de la nueva clase en la matriz de confusión.

5. **(15%)** Implemente manualmente el clasificador k-NN, sin utilizar `sklearn`:

   `sklearn.neighbors.KNeighborsClassifier`,

   completando la función `knn_passo_a_passo` presentada en el capítulo. Compare la exactitud y el tiempo de ejecución de la implementación manual con la implementación de `scikit-learn` en conjuntos de datos de tamaños crecientes y relacione los resultados con la maldición de la dimensionalidad.

6. **(15%)** Investigue la influencia de los parámetros `orientations`, `pixels_per_cell` y `cells_per_block` del descriptor HOG en la base `load_digits`. Evalúe al menos cuatro combinaciones de parámetros y discuta el compromiso entre dimensionalidad del descriptor y desempeño del clasificador.

7. **(15%)** Evalúe la influencia de los parámetros P (número de vecinos) y R (radio) del descriptor LBP en la clasificación de las texturas sintéticas del Proyecto Práctico 2, considerando $P \in \{4, 8, 16\}$ y $R \in \{1, 2, 3\}$. Analice cómo estos parámetros afectan la capacidad discriminativa del descriptor.

8. **(Bono – 10%)** Implemente manualmente la validación cruzada *k-fold* para el clasificador k-NN en la base `load_digits`, sin utilizar `cross_val_score`, y compare los resultados con los obtenidos por la implementación de `scikit-learn` presentada en la Figura 7.16.

Referencias del Capítulo

La fundamentación teórica y los experimentos presentados en este capítulo se basan en las siguientes referencias:

- Gonzalez (2018), por los fundamentos de descriptores estadísticos de textura y de las operaciones de preprocesamiento aplicadas a la extracción de características.
- Szeliski (2022), por la presentación del pipeline clásico de reconocimiento de patrones, de la extracción de descriptores y de la evaluación de clasificadores en Visión por Computadora.
- Duda (2001), por los fundamentos teóricos del reconocimiento de patrones, del clasificador k-NN y de la relación entre sesgo y varianza.
- Cover (1967), por la formulación original del algoritmo de los k vecinos más cercanos.
- Ojala (2002), por la formulación del descriptor *Local Binary Patterns* (LBP) y de su variante uniforme, utilizada en este capítulo.
- Dalal (2005), por la formulación del descriptor *Histogram of Oriented Gradients* (HOG), empleado en la representación de forma y contorno.
- Pedregosa (2011), por la implementación del clasificador k-NN, de las métricas de evaluación y de la validación cruzada en la biblioteca `scikit-learn`.
- Quilici-gonzalez (2014), por la presentación didáctica de clasificadores tradicionales de Reconocimiento de Patrones, como Árboles de Decisión, Reglas de Clasificación y Máquinas de Vectores de Soporte (SVM), complementarios al clasificador k-NN explorado en este capítulo.
- Quilici-gonzalez (2026), por la actualización y ampliación de estos contenidos en su 2ª edición, actualmente en producción.

Executar Colab | Abrir GitHub

7.18 💻 Parte Práctica con Ejercicios de Programación

La presente lista de ejercicios de programación (EP) consolida las formulaciones teóricas presentadas a lo largo del Capítulo 7 — Clasificación de Imágenes y Reconocimiento de Patrones — mediante una ruta práctica aplicada. A diferencia de la manipulación directa de píxeles de los capítulos anteriores, los EPs de este capítulo trabajan con las **magnitudes intermedias** de un *pipeline* real de reconocimiento de patrones — vectores de características, distancias, etiquetas previstas y reales, códigos binarios locales e histogramas de orientación — permitiendo validar manualmente cada etapa del razonamiento sin depender de bibliotecas externas de aprendizaje automático.

El encadenamiento de los ejercicios reproduce el flujo conceptual del capítulo: se comienza con la implementación manual de la regla de decisión del clasificador **k-NN** sobre un pequeño espacio de características; a continuación, se revisita, desde la óptica de la normalización de características, el clasificador implementado en el primer ejercicio de la lista; se avanza hacia el cálculo de las métricas de **evaluación** (matriz de confusión, precisión y exhaustividad) a partir de etiquetas previstas y reales; se prosigue con la codificación manual del descriptor de textura **LBP** a partir de una vecindad 3×3; se profundiza en el cálculo del histograma de orientaciones del descriptor **HOG** para una única celda; se avanza, a continuación, hacia la integración de **extracción de descriptores**, **clasificación k-NN** y **evaluación multiclase** en un *pipeline* completo de reconocimiento de texturas; y se concluye con la aplicación de ese mismo *pipeline* sobre una **imagen real** (formato PGM), en el que el descriptor LBP se calcula directamente sobre los píxeles de un mosaico de texturas.

🎯 Objetivo de este Cuaderno

El cuaderno permite desarrollar, validar, organizar y probar soluciones de **Ejercicios de Programación (EPs)** en entornos interactivos, como Colab, con los mismos casos de prueba de Moodle, copiándolos allí solo al momento de registrar la nota oficial.

Download

Descargue `morph.py` y `testsuite.py` ejecutando la celda siguiente:

```
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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Ejecutando las pruebas

Para evaluar las pruebas, ejecute `TestSuite("EP07_01.extensión").run()` en una nueva celda, reemplazando la extensión por la del lenguaje utilizado (`.py`, `.java`, `.c`, `.cpp`, `.js` o `.r`). El sistema descarga los casos de prueba de GitHub, ejecuta el programa y calcula la nota automáticamente.

Para probar código Python directamente, sin guardar archivo, use `run_code(codigo)` pasando el código como *string* en una variable `codigo`:

```
codigo = """
# ... su código aquí ...
"""
TestSuite("EP07_01").run_code(codigo)
```

Resumen de los Métodos de `morph.py` (Cap. 7)

La biblioteca `morph.py` ofrece dos versiones para la mayoría de los algoritmos: una **didáctica** (métodos terminados en `0`), implementada paso a paso en NumPy, y otra **clásica**, basada en las bibliotecas `scikit-learn` y `scikit-image`. Las implementaciones didácticas se utilizan en los **Ejercicios de Programación (EPs)**, ya que no dependen de bibliotecas externas y se ejecutan dentro del límite de memoria del entorno **VPL** de Moodle. En cambio, las versiones clásicas son más eficientes y se recomiendan para experimentos en entornos como Colab y Jupyter Notebook, pero normalmente **no pueden utilizarse en los EPs** de Moodle, ya que la biblioteca `scikit-learn` excede la memoria disponible en el VPL.

1. **Lectura de datos (`readClasses`, `readDataset`, `readTrain`, `readTest`)**
   Estandarizan la entrada de los conjuntos de entrenamiento y prueba, devolviendo las matrices de características (X) y los vectores de etiquetas (y).

2. **Clasificación (`knn0` / `knn`)**
   Implementan el algoritmo de los **k-vecinos más cercanos (k-NN)** para clasificación binaria y multiclase, utilizando distancia Euclidiana o Manhattan.

3. **Normalización (`zscore0` / `zscore`)**
   Aplican la normalización *z-score* a los atributos, reduciendo diferencias de escala antes de la clasificación.

4. **Evaluación (`confusion0` / `confusion`)**
   Calculan la matriz de confusión y métricas como exactitud, precisión y recuperación, tanto para problemas binarios como multiclase.

5. **Descriptor de textura (`lbp0` / `lbp`)**
   Calculan el ***Local Binary Pattern*** **(LBP)**, permitiendo obtener el mapa LBP, el código de un píxel o el histograma de una región de la imagen.

6. **Descriptor de forma (`hog0` / `hog`)**
   Calculan el ***Histogram of Oriented Gradients*** **(HOG)**, produciendo histogramas de las orientaciones de los gradientes para representar información de forma y contorno.

7.18.1 EP07_01 🟢 Clasificador k-NN Paso a Paso

El `KNeighborsClassifier` de `scikit-learn`, utilizado a lo largo del capítulo, oculta detrás de una única llamada (`.fit` / `.predict`) una regla de decisión bastante simple: para cada nueva observación, calcular la distancia a todos los ejemplos de entrenamiento, seleccionar los k más cercanos y votar por la clase mayoritaria entre ellos.

Antes de confiar en la biblioteca, se te ha encargado implementar esta regla desde cero, para un espacio de características bidimensional, exactamente como el simulador interactivo de frontera de decisión del capítulo lo hace internamente en cada clic del usuario.

7.18.1.1 📋 Directrices de Implementación

1. **Cantidad y parámetro:** Leer el entero N (número de ejemplos de entrenamiento) y el entero impar k (número de vecinos).
2. **Ejemplos de entrenamiento:** Para cada uno de los N ejemplos, leer tres valores: las coordenadas x e y (reales) y la etiqueta r (entera, 0 o 1).
3. **Consultas:** Leer el entero Q (número de puntos de consulta) y, a continuación, las coordenadas x_q, y_q (reales) de cada consulta.

4. **Distancia:** Para cada consulta, calcular la distancia euclidiana hasta **todos** los ejemplos de entrenamiento:
$$d(x_q, x_i) = \sqrt{(x_q - x_i)^2 + (y_q - y_i)^2}.$$
5. **Selección de vecinos:** Ordenar los ejemplos por distancia creciente y seleccionar los k primeros. En caso de **empate de distancia** en la frontera del k-ésimo vecino, desempatar por el ejemplo leído **primero** en la entrada (orden de lectura estable).
6. **Votación mayoritaria:** Contar los votos de cada clase entre los k vecinos seleccionados. Si hay **empate en la votación** (solo posible cuando k es par, lo que no debería ocurrir según la directriz del punto 1, pero tratar defensivamente), asignar la clase del vecino más cercano entre las clases empatadas.
7. **Salida:** Para cada consulta, en el orden de entrada, imprimir la clase predicha. Al final, imprimir el total de consultas clasificadas como clase `1`.

7.18.1.2 📌 Restricciones Computacionales

- **Métrica fija:** utilizar exclusivamente la distancia euclidiana (no la *distancia al cuadrado*) para la ordenación, aunque el resultado de la comparación sea el mismo.
- **k siempre impar:** la entrada garantiza k impar y $k \leq N$; aun así, implementar el desempate del punto 6 por robustez.
- **Estabilidad:** al ordenar por distancia, preservar el orden relativo de ejemplos con la misma distancia (ordenación estable).

7.18.1.3 🧠 Fundamentación Teórica

Elemento	Papel en el k-NN
Espacio de características	Conjunto de todos los vectores (x, y) posibles
Distancia euclidiana	Medida de similitud entre observaciones
k pequeño	Frontera irregular, alta varianza
k grande	Frontera suave, alto sesgo
Votación mayoritaria	Regla de decisión $\hat{y} = \text{moda}\{y_i : x_i \in N_k(x)\}$

7.18.1.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

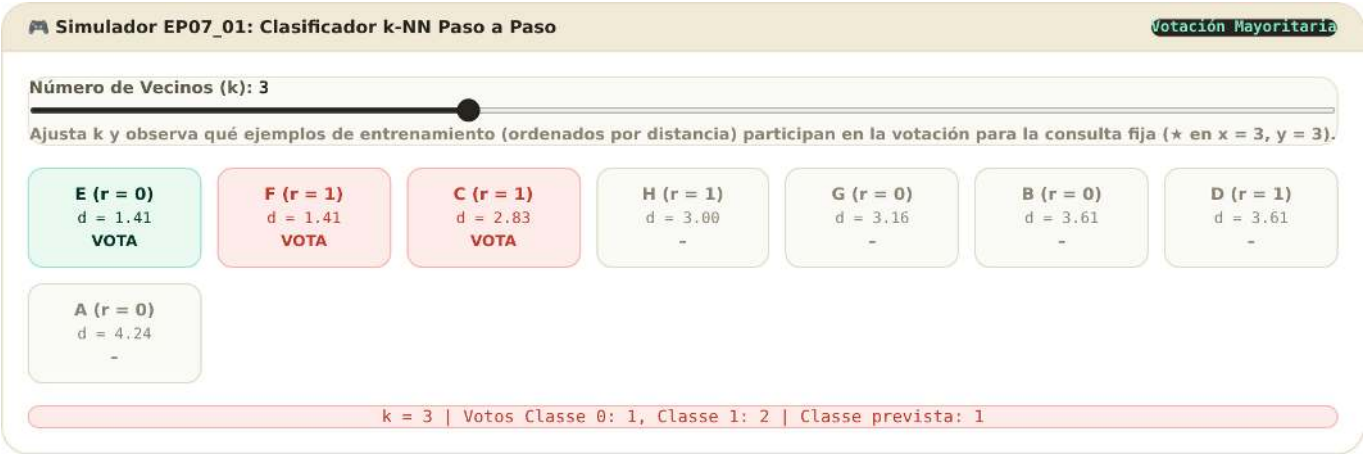
- Línea 1: Enteros N y k, separados por espacio.
- Siguientes N líneas: tres valores por línea — x, y (reales) y r (entero $\in \{0, 1\}$), separados por espacio.
- Siguiente línea: entero Q.
- Siguientes Q líneas: dos valores por línea — x_q, y_q (reales), separados por espacio.

Salida:

- Q líneas, cada una con la clase predicha (`0` o `1`) para la respectiva consulta, en el orden de entrada.
- Última línea: `Total clase 1: X`.

7.18.1.5 📌 Ejemplos

Entrada	Salida	Observación
4 3 0 0 0 1 0 0 5 5 1 6 5 1 1 1 1	0 Total clase 1: 0	Consulta cercana al agrupamiento de clase 0.
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 clase 1: 1	Con $k = 1$, cada consulta hereda la clase del vecino más cercano.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap07/sim-ep0701.html

Figura 7.21: Simulador EP07_01: Clasificador k-NN Paso a Paso

```
%%writefile EP07_01.py
# Código Python
```

```
Overwriting EP07_01.py
```

```
TestSuite("EP07_01.py").run()
```

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

7.18.2 EP07_02 🟡 Normalización *Z-score* y Robustez del k-NN ante Escalas Distintas

Este ejercicio revisita el clasificador implementado en el **EP07_01**, esta vez bajo la óptica discutida en la sección *El Impacto de la Escala y la Normalización de Características* del capítulo: el k-NN decide basándose en la distancia entre vectores, de modo que una característica medida en una escala mucho mayor que las demás tiende a **dominar** el cálculo de la distancia, incluso cuando no es la más relevante para separar las clases.

Un sistema de inspección registra, para cada pieza, su **área** (en píxeles, pudiendo llegar a cientos o miles) y su **circularidad** (siempre entre 0 y 1). Usted ha sido encargado de clasificar nuevas piezas mediante k-NN

de dos formas — con y sin la estandarización *Z-score* presentada en el capítulo — y de reportar en qué casos ambas aproximaciones **divergen**.

7.18.2.1 📋 Directrices de Implementación

1. **Cantidad y parámetro:** Leer el entero N (número de ejemplos de entrenamiento) y el entero impar k.
2. **Ejemplos de entrenamiento:** Para cada uno de los N ejemplos, leer tres valores: el área x_1 (real), la circularidad x_2 (real) y la etiqueta r (entero, 0 o 1).
3. **Consultas:** Leer el entero Q y, a continuación, las coordenadas x_1, x_2 de cada consulta.
4. **Clasificación sin normalización:** Para cada consulta, clasifíquela mediante k-NN directamente sobre (x_1, x_2), con distancia euclidiana y las mismas reglas de desempate del EP07_01 (orden de lectura para distancias empatadas; vecino más cercano entre clases empatadas en la votación).
5. **Parámetros de normalización:** Calcular la media μ_j y la desviación estándar **poblacional** σ_j (división por N, no por $N-1$ — la misma convención adoptada por la clase `StandardScaler`) de cada característica $j \in \{1, 2\}$, **exclusivamente sobre el conjunto de entrenamiento**.
6. **Estandarización:** Transformar cada característica de entrenamiento y de consulta mediante

$$z_j = \frac{x_j - \mu_j}{\sigma_j}.$$

   Si $\sigma_j = 0$ (característica constante en el entrenamiento), definir $z_j = 0$ para todas las muestras de esa característica, evitando la división por cero.
7. **Clasificación con normalización:** Repetir la clasificación k-NN del punto 4, ahora sobre los vectores estandarizados (z_1, z_2), con las mismas reglas de desempate.
8. **Salida:** Para cada consulta, en el orden de entrada, imprimir las dos clases previstas. Al final, imprimir el número de consultas en las que las dos clasificaciones **divergen**.

7.18.2.2 📌 Restricciones Computacionales

- **Ajuste solo en el entrenamiento:** μ_j y σ_j se calculan únicamente a partir del conjunto de entrenamiento y se reaplican a las consultas — nunca se recalculan a partir de ellas. Esta práctica evita la **fuga de datos** (*data leakage*), mencionada en la sección de normalización del capítulo.
- **Desviación estándar poblacional:** utilizar $\sigma_j = \sqrt{\frac{1}{N}\sum_i (x_{i,j} - \mu_j)^2}$, y no la versión muestral (división por $N-1$).
- **Característica constante:** tratar $\sigma_j = 0$ como caso especial (punto 6); no debe ocurrir un error de división por cero.
- **Reglas de desempate:** reutilizar exactamente las convenciones del EP07_01, tanto en la selección de los k vecinos como en la votación mayoritaria.

7.18.2.3 🧠 Fundamentación Teórica

Elemento	Papel
Estandarización *Z-score*	Reescala cada característica para media 0 y desviación estándar 1, haciendo comparables escalas heterogéneas
Ajuste (*fit*) solo en el entrenamiento	Garantiza que la evaluación sobre las consultas refleje únicamente lo que el modelo aprendió en el entrenamiento
Distancia euclidiana sin normalización	Dominada por la característica de mayor amplitud — aquí, el área
Predicción divergente	Evidencia que la escala de las características, y no solo el algoritmo o los datos, puede determinar la frontera de decisión del k-NN

Este ejercicio refuerza, de forma controlada, la razón por la cual el `StandardScaler` se aplica antes del k-NN a lo largo del capítulo: sin esta etapa, las características de circularidad — incluso siendo altamente discriminativas — pueden ser prácticamente ignoradas por el clasificador frente a una característica de área con amplitud cientos de veces mayor.

7.18.2.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

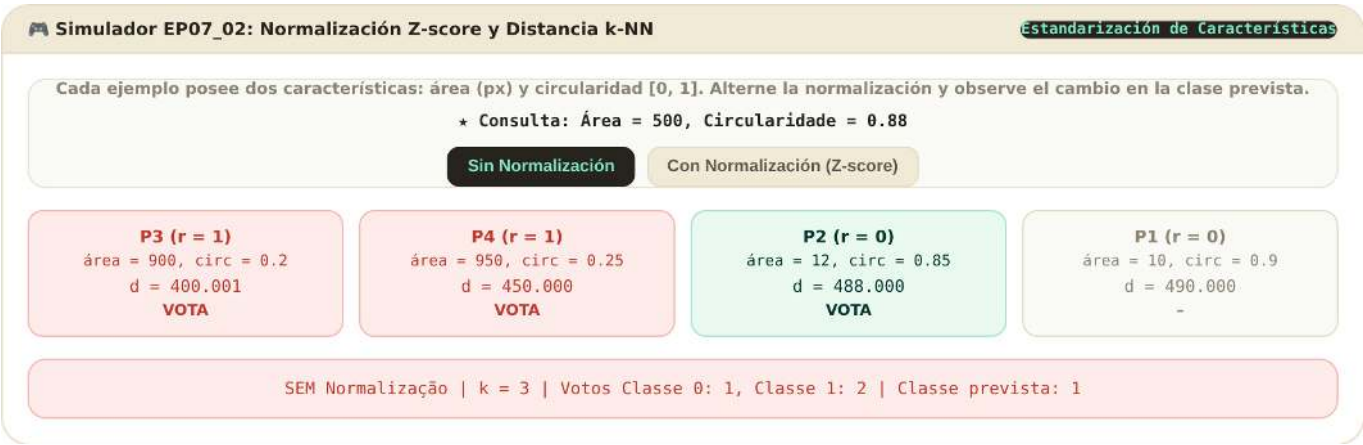
- Línea 1: Enteros N y k, separados por espacio.
- Siguientes N líneas: tres valores por línea — x_1, x_2 (reales) y r (entero $\in \{0, 1\}$), separados por espacio.
- Siguiente línea: entero Q.
- Siguientes Q líneas: dos valores por línea — x_1, x_2 (reales) de la consulta, separados por espacio.

Salida:

- Q líneas, en el formato `SemNorm=<0|1> ComNorm=<0|1>`, en el orden de entrada de las consultas.
- Última línea: `Divergiu: <int>`.

7.18.2.5 📌 Ejemplos

Entrada	Salida	Observación
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	Sin normalización, el área (escala de cientos) domina la distancia y la consulta se clasifica como clase `1`. Tras la estandarización, la circularidad — mucho más cercana a las muestras de clase `0` — pasa a pesar de forma comparable, y la predicción cambia a `0`.
2 1 0 0.5 0 100 0.5 1 1 60 0.5	SemNorm=1 ComNorm=1 Divergiu: 0	La circularidad es constante en el entrenamiento ($\sigma_2 = 0$); por la regla del punto 6, $z_2 = 0$ para todas las muestras, y la clasificación depende solo del área en ambos casos.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap07/sim-ep0702.html

Figura 7.22: Simulador EP07_02: Efecto de la Normalización *Z-score* en la Distancia k-NN

```
%%writefile EP07_02.py
# Código Python
```

```
Overwriting EP07_02.py
```

```
TestSuite("EP07_02.py").run()
```

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

7.18.3 EP07_03 🟡 Evaluación por Matriz de Confusión

Un clasificador binario de calidad de soldadura fue entrenado y probado en una línea de producción. Para cada pieza inspeccionada, el sistema registró la etiqueta **real** (obtenida por un especialista) y la etiqueta **predicha** por el clasificador, donde `1` representa "defectuosa" y `0` representa "conforme".

La gerencia de calidad quiere saber no solo la exactitud del sistema, sino también su **precisión** (cuando el sistema señala un defecto, ¿con qué frecuencia está en lo correcto?) y su **revocación** (de todas las piezas realmente defectuosas, ¿cuántas logró identificar el sistema?) — la distinción discutida en la sección de evaluación de clasificadores del capítulo.

7.18.3.1 📋 Directrices de Implementación

1. **Cantidad:** Leer el entero N (número de piezas inspeccionadas).
2. **Datos de cada pieza:** Para cada una de las N piezas, leer dos enteros — la etiqueta real y y la etiqueta predicha $\hat{y}$ (ambas $\in \{0, 1\}$).
3. **Matriz de confusión:** Considerando la clase `1` (defectuosa) como **positiva**, contar:
   - VP (Verdadero Positivo): $y = 1$ y $\hat{y} = 1$;
   - FP (Falso Positivo): $y = 0$ y $\hat{y} = 1$;
   - FN (Falso Negativo): $y = 1$ y $\hat{y} = 0$;
   - VN (Verdadero Negativo): $y = 0$ y $\hat{y} = 0$.
4. **Métricas:** Calcular
$$\text{Exactitud} = \frac{VP + VN}{N}, \quad \text{Precisión} = \frac{VP}{VP + FP}, \quad \text{Revocación} = \frac{VP}{VP + FN}.$$
5. **Casos degenerados:** Si $VP+FP = 0$ (ninguna predicción positiva), imprimir `Precisao: indefinida`. Si $VP + FN = 0$ (ningún caso positivo real), imprimir `Revocacao: indefinida`.
6. **Redondeo:** Todas las métricas numéricas deben redondearse a 4 decimales (*redondeo half away from zero*) solo en la visualización.

7.18.3.2 📌 Restricciones Computacionales

- **Convención de clase positiva fija:** la clase `1` es siempre la clase positiva en este ejercicio, independientemente de su frecuencia relativa.
- **Protección contra división por cero:** implemente los casos degenerados del punto 5 antes de realizar la división.
- **Orden de salida:** siga exactamente el orden especificado en la sección de salida, incluso en los casos degenerados.

7.18.3.3 🧠 Fundamentación Teórica

Métrica	Pregunta que responde	¿Sensible a desbalanceo?
Exactitud	¿Qué fracción de las piezas fue clasificada correctamente?	Sí — puede enmascarar errores en la clase minoritaria

Métrica	Pregunta que responde	¿Sensible a desbalanceo?
Precisión	De las piezas señaladas como defectuosas, ¿cuántas realmente lo son?	Penaliza falsos positivos
Revocación	De las piezas realmente defectuosas, ¿cuántas fueron detectadas?	Penaliza falsos negativos

En un contexto industrial, una **revocación** baja es frecuentemente más grave que una **precisión** baja: dejar pasar una pieza defectuosa (falso negativo) tiende a ser más costoso que inspeccionar manualmente una pieza buena señalada por error (falso positivo).

7.18.3.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

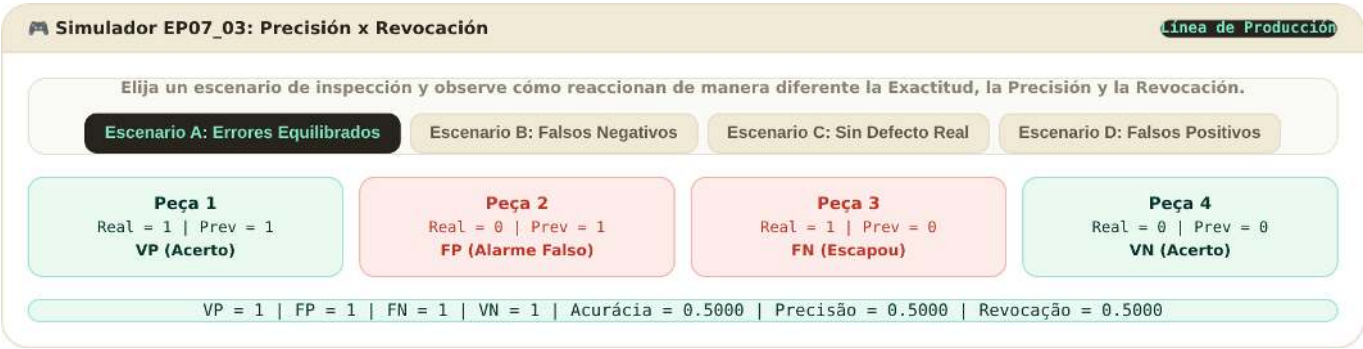
- Línea 1: Entero N.
- Siguientes N líneas: dos enteros por línea — y y $\hat{y}$, separados por espacio.

Salida (en este orden exacto):

```
VP=<int> FP=<int> FN=<int> VN=<int>
Acuracia: <valor o métrica indefinida>
Precisao: <valor o indefinida>
Revocacao: <valor o indefinida>
```

7.18.3.5 📌 Ejemplos

Entrada	Salida	Observación
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	Un error de cada tipo.
3 0 0 0 0 0 0	VP=0 FP=0 FN=0 VN=3 Acuracia: 1.0000 Precisao: indefinida Revocacao: indefinida	Ningún caso positivo real ni predicho.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap07/sim-ep0703.html

Figura 7.23: Simulador EP07_03: Precisión x Revocación

```
%%writefile EP07_03.py
# Código Python
```

```
Overwriting EP07_03.py
```

```
TestSuite("EP07_03.py").run()
```

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

7.18.4 EP07_04 🟠 Codificación Manual del Descriptor LBP

La función `local_binary_pattern` de `scikit-image`, utilizada en el proyecto de clasificación de texturas, calcula automáticamente el código LBP de cada píxel de una imagen. Antes de utilizarla como una caja negra, se le ha encargado implementar manualmente el cálculo del código LBP clásico ($P = 8$, $R = 1$) para el píxel central de una vecindad 3×3, exactamente como se define en la ecuación del capítulo.

Además del código, el sistema de inspección de texturas también necesita saber si ese patrón es **uniforme** — un patrón es uniforme cuando el número de transiciones ($0 \to 1$ o $1 \to 0$) al recorrer los 8 bits **circularmente** (volviendo del último bit al primero) es **como máximo 2**, propiedad explorada por la variante *uniforme* del LBP mencionada en el capítulo.

7.18.4.1 📋 Directrices de Implementación

1. **Cantidad:** Leer el entero T (número de vecindades a procesar).
2. **Datos de cada vecindad:** Para cada una de las T vecindades, leer una matriz 3×3 de enteros (intensidades), proporcionada en 3 líneas de 3 valores cada una. El píxel central es la posición `[1][1]`.
3. **Orden de los vecinos:** Recorra los 8 vecinos en sentido **horario**, comenzando en la esquina superior izquierda, en el siguiente orden de posiciones `[fila][columna]`: `[0][0]`, `[0][1]`, `[0][2]`, `[1][2]`, `[2][2]`, `[2][1]`, `[2][0]`, `[1][0]`. Este es el índice $p = 0, 1, \ldots, 7$ de la ecuación del LBP.
4. **Función umbral:** Para cada vecino p con intensidad g_p y centro g_c, calcule $s(g_p - g_c)$, que vale `1` si $g_p \geq g_c$ y `0` en caso contrario.
5. **Código LBP:** Calcule
$$\mathrm{LBP} = \sum_{p=0}^{7} s(g_p - g_c)\, 2^p.$$
6. **Transiciones:** Considerando la secuencia circular de bits $s_0, s_1, \ldots, s_7$ (en el orden del punto 3), cuente cuántos pares consecutivos **adyacentes en la secuencia circular** (incluyendo el par s_7, s_0) difieren entre sí.
7. **Clasificación:** Si el número de transiciones es ≤ 2, clasifique como `UNIFORME`; en caso contrario, `NAO_UNIFORME`.
8. **Salida:** Para cada vecindad, en el orden de entrada, imprimir el código LBP (entero decimal, 0–255), el número de transiciones y la clasificación.

7.18.4.2 📌 Restricciones Computacionales

- **Orden fija de los vecinos:** el orden del punto 3 es obligatorio — invertirlo produce un código numéricamente diferente, incluso representando el mismo patrón visual.
- **Comparación no estricta:** $s(z) = 1$ cuando $z \geq 0$ (el propio capítulo define la igualdad como incluida en el caso `1`).
- **Conteo circular:** no olvide el par que cierra el ciclo (s_7 con s_0); ignorar ese par es un error común que clasifica incorrectamente patrones uniformes.

7.18.4.3 🧠 Fundamentación Teórica

Patrón (bits $s_0 \dots s_7$)	Transiciones	Interpretación
`00000000` o `11111111`	0	Región homogénea (mancha clara u oscura)
`00001111`	2	Borde simple entre dos regiones
`01010101`	8	Textura de contraste alternado — no uniforme

Los patrones uniformes se concentran en regiones de textura suave o bordes simples; los patrones no uniformes tienden a corresponder a ruido de alta frecuencia. Por eso, el histograma LBP *uniforme*, usado en el proyecto de clasificación de texturas, agrupa todos los patrones no uniformes en un único compartimento, reduciendo la dimensionalidad del descriptor.

7.18.4.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero T.
- Para cada vecindad: 3 líneas con 3 enteros cada una (matriz 3×3).

Salida:

- T líneas, en el formato `LBP=<int> transicoes=<int> <UNIFORME|NAO_UNIFORME>`.

7.18.4.5 📌 Ejemplos

Entrada	Salida	Observación
1 10 10 10 10 50 10 10 10 10	LBP=0 transicoes=0 UNIFORME	El centro es el más claro; todos los vecinos generan bit 0.
1 90 90 90 10 50 10 90 90 90	LBP=119 transicoes=4 NAO_UNIFORME	Vecinos claros y oscuros alternados en la vecindad.

```
%%writefile EP07_04.py
# Código Python
```

```
Overwriting EP07_04.py
```

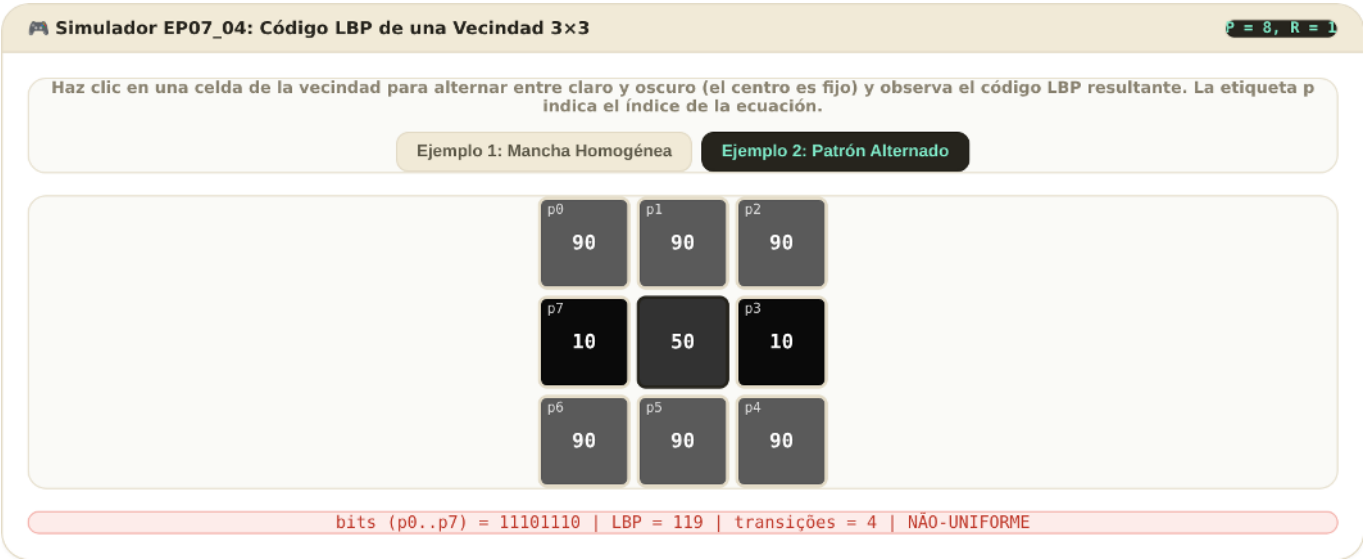
```
TestSuite("EP07_04.py").run()
```

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

7.18.5 EP07_05 🔴 Histograma de Orientaciones de una Célula HOG

La función `hog` de `scikit-image`, empleada en el proyecto de clasificación de dígitos, divide la imagen en pequeñas **células** y, para cada una, construye un histograma de las orientaciones del gradiente ponderado por la magnitud — exactamente la etapa central descrita en la sección sobre el descriptor HOG del capítulo.

Se te ha encargado implementar este cálculo para una única célula, a partir de los valores de magnitud y orientación del gradiente **ya calculados** para cada píxel de la célula (prescindiendo del cálculo de las derivadas parciales).



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap07/sim-ep0704.html

Figura 7.24: Simulador EP07_04: Código LBP de una Vecindad 3×3

7.18.5.1 📋 Directrices de Implementación

1. **Dimensiones:** Leer los enteros n (la célula tiene $n \times n$ píxeles) y B (número de compartimentos del histograma).
2. **Magnitudes:** Leer n líneas con n valores reales cada una, que representan $|\nabla f(x, y)|$ para cada píxel de la célula.
3. **Orientaciones:** Leer otras n líneas con n valores reales cada una, que representan $\theta(x, y)$ en **grados**, ya convertidos al intervalo **no signado** $[0°, 180°)$, como se utiliza convencionalmente en HOG.
4. **Compartimentos:** Los B compartimentos cubren $[0°, 180°)$ en bandas iguales de ancho $180/B$ grados. Un píxel con orientación θ pertenece al compartimento $\lfloor \theta/(180/B) \rfloor$; si ese índice es igual a B (posible solo cuando θ es exactamente 180°, lo cual no debería ocurrir según la directriz del punto 3), utiliza el compartimento $B - 1$.
5. **Histograma bruto:** Para cada píxel, acumula su **magnitud** (no su conteo) en el compartimento correspondiente:
$$H[b] = \sum_{(x,y)\,:\,\mathrm{bin}(\theta(x,y))=b} |\nabla f(x, y)|.$$
6. **Normalización L2:** Tras construir H, normalízalo para obtener $\hat{H}$:
$$\hat{H}[b] = \frac{H[b]}{\sqrt{\sum_{j=0}^{B-1} H[j]^2 + \epsilon}}, \qquad \epsilon = 10^{-6}.$$
7. **Salida:** Imprimir el histograma bruto H (redondeado a 2 decimales) en una línea, seguido del histograma normalizado $\hat{H}$ (redondeado a 4 decimales) en otra línea, ambos con los B valores separados por espacios, en el orden de los compartimentos.

7.18.5.2 📌 Restricciones Computacionales

- ***Binning* no signado:** el intervalo de orientaciones es $[0, 180)$, no $[0, 360)$ — los gradientes en direcciones opuestas (diferencia de 180°) contribuyen al **mismo** compartimento, convención estándar de HOG para detección de objetos.
- **Acumulación por magnitud, no por conteo:** el histograma pondera cada píxel por su magnitud de gradiente, no simplemente cuenta cuántos píxeles caen en cada compartimento.
- **Constante de estabilización:** el $\epsilon = 10^{-6}$ en el denominador de la normalización evita la división por cero cuando la célula es completamente homogénea (todas las magnitudes nulas).

7.18.5.3 📐 De dónde provienen las matrices de entrada

Antes de este EP, cada píxel (x, y) de la imagen pasa por:

$$G_x = f(x+1, y) - f(x-1, y), \qquad G_y = f(x, y+1) - f(x, y-1)$$

$$|\nabla f| = \sqrt{G_x^2 + G_y^2}, \qquad \theta_{\text{signado}} = \text{atan2}(G_y, G_x)$$

Como HOG ignora la polaridad del contraste, el ángulo se pliega al intervalo no signado:

$$\theta = \theta_{\text{signado}} \bmod 180^\circ$$

Repitiendo esto para todos los píxeles de una célula $n \times n$, se obtienen las dos matrices de entrada de este ejercicio: **magnitudes** $|\nabla f|$ y **orientaciones** $\theta \in [0^\circ, 180^\circ)$.

7.18.5.4 🧠 Fundamentación Teórica

Etapa	Papel
Magnitud del gradiente	Pondera la contribución de cada píxel — los bordes fuertes pesan más que el ruido débil
Orientación no signada	Hace que el descriptor sea invariante a la polaridad del contraste (claro→oscuro vs. oscuro→claro)
Histograma por célula	Resume la distribución local de bordes en un vector compacto
Normalización L2	Reduce la sensibilidad del descriptor a variaciones globales de iluminación y contraste

La concatenación de los histogramas normalizados de todas las células de la imagen — no implementada en este ejercicio — forma el vector de características HOG completo, utilizado como entrada del clasificador k-NN en el proyecto del capítulo.

7.18.5.5 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Enteros n y B.
- Siguientes n líneas: n magnitudes reales cada una.
- Siguientes n líneas: n orientaciones reales (grados, $[0, 180)$) cada una.

Salida:

- Línea 1: los B valores del histograma bruto, redondeados a 2 decimales.
- Línea 2: los B valores del histograma normalizado, redondeados a 4 decimales.

7.18.5.6 📌 Ejemplos

Entrada	Salida	Observación
2 2 1.0 2.0 3.0 4.0 10 100 170 20	5.00 5.00 0.7071 0.7071	Bin de ancho 90°: $[0, 90)$ y $[90, 180)$; magnitudes 1 y 4 caen en el bin 0, 2 y 3 en el 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	Célula homogénea: ϵ evita división por cero.

```
%%writefile EP07_05.py
# Código Python
```

```
Overwriting EP07_05.py
```

```
TestSuite("EP07_05.py").run()
```

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

7.18.6 EP07_06 🟣 *Pipeline* Completo: Descriptores + k-NN + Evaluación Multiclase

Este ejercicio integra las tres etapas centrales del capítulo en un único *pipeline*, reproduciendo en miniatura el **Proyecto Práctico 2** (clasificación de texturas sintéticas por LBP): un conjunto de histogramas de descriptores **ya extraídos** (como si fueran histogramas LBP) se utiliza para entrenar un clasificador k-NN, que a su vez se evalúa sobre un conjunto de prueba independiente mediante una matriz de confusión multiclase.

A diferencia de EP07_01, aquí el espacio de características tiene dimensión arbitraria H (el tamaño del histograma), existen más de dos clases, y la métrica de distancia es un parámetro de entrada — lo que permite reproducir el experimento de comparación de métricas discutido en el capítulo.

7.18.6.1 📋 Directrices de Implementación

1. **Clases:** Leer el entero C (número de clases) seguido de C nombres de clase (*strings* sin espacio), en el orden en que deben aparecer en la matriz de confusión.
2. **Configuración:** Leer el entero H (dimensión de los histogramas), la *string* M (métrica: `euclidiana` o `manhattan`) y el entero impar k.
3. **Entrenamiento:** Leer el entero N y, a continuación, N líneas, cada una conteniendo el nombre de la clase seguido de H valores reales (el histograma de descriptor).
4. **Prueba:** Leer el entero Q y, a continuación, Q líneas, cada una conteniendo el nombre de la clase **real** seguido de H valores reales (el histograma de descriptor de la muestra de prueba).
5. **Distancia:** Para cada muestra de prueba, calcule la distancia a cada ejemplo de entrenamiento usando la métrica 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. **Clasificación k-NN:** Seleccione los k ejemplos de entrenamiento más cercanos (desempate de distancia por el orden de lectura, como en EP07_01) y clasifique por la clase mayoritaria entre ellos. En caso de **empate de votación** entre dos o más clases, elija la que aparezca **primero** en la lista de clases del ítem 1.
7. **Matriz de confusión:** Construya una matriz $C \times C$ en la que la fila corresponde a la clase real y la columna a la clase prevista, siguiendo el orden de clases del ítem 1.

Simulador EP07_05: Histograma de Orientaciones de una Célula célula 3×3 fija

Ajusta B y observa cómo la **matriz de orientaciones** (independiente de la de magnitudes) se mapea a los compartimentos mediante `bin = floor(θ / (180/B))`, y cómo las magnitudes se suman en cada bin.

Número de compartimentos (B) 2

Entrada (exactamente como el programa lee por 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
```

Matriz de magnitudes |∇f|

1.0	2.0	0.5
3.0	4.0	1.5
0.8	2.5	3.2

Matriz de orientaciones θ (grados) — coloreada por el bin

10° ⌊·⌋=0.11 ⌊·⌋=bin 0	100° ⌊·⌋=1.11 ⌊·⌋=bin 1	45° ⌊·⌋=0.50 ⌊·⌋=bin 0
170° ⌊·⌋=1.89 ⌊·⌋=bin 1	20° ⌊·⌋=0.22 ⌊·⌋=bin 0	95° ⌊·⌋=1.06 ⌊·⌋=bin 1
60° ⌊·⌋=0.67 ⌊·⌋=bin 0	150° ⌊·⌋=1.67 ⌊·⌋=bin 1	5° ⌊·⌋=0.06 ⌊·⌋=bin 0

Dónde cae cada θ en la regla [0°, 180°) — bin = floor(θ / ancho)

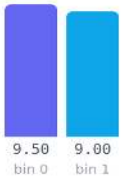
Rango de cada compartimento (ancho = 180° / B)

bin 0: [0.0°, 90.0°) bin 1: [90.0°, 180.0°)

Cada píxel: magnitud + orientación → 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

Histograma bruto (H) | Histograma normalizado (Ĥ)



Cálculo paso a paso (floor de la división + suma de magnitudes por 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.es/cap07/sim-ep0705.html

Figura 7.25: Simulador EP07_05: Histograma HOG de una Celda (mapeo de ángulos a bins)

8. **Precisión:** Calcule la precisión global como la razón entre aciertos y Q.
9. **Salida:** Para cada muestra de prueba, en el orden de entrada, imprimir la clase prevista. A continuación, imprimir la matriz de confusión (una fila por clase real, valores separados por espacio, en el orden de las clases). Finalmente, imprimir la precisión redondeada a 4 decimales.

7.18.6.2 📌 Restricciones Computacionales

- **Métrica seleccionable:** implemente ambas distancias; la métrica M define cuál se utiliza en toda la ejecución (no es posible mezclar métricas en la misma llamada).
- **Desempate de votación determinístico:** el criterio del ítem 6 (orden de la lista de clases) debe seguirse incluso cuando el empate involucra más de dos clases.
- **Independencia de entrenamiento y prueba:** no hay necesidad de validar que las muestras de prueba no aparecen en el entrenamiento — asuma que la entrada es válida.

7.18.6.3 🧠 Fundamentación Teórica

Etapa del ejercicio	Etapa correspondiente en el capítulo
Histogramas de entrenamiento/prueba ya extraídos	`descritor_lbp` aplicado a las texturas sintéticas
Distancia euclidiana o Manhattan	Parámetro `metric` del `KNeighborsClassifier`
Votación mayoritaria con k vecinos	`KNeighborsClassifier.predict`
Matriz de confusión $C \times C$	`confusion_matrix` de `scikit-learn`
Precisión global	`accuracy_score` de `scikit-learn`

Este ejercicio evidencia, de forma controlada, un resultado discutido en el capítulo: la **elección de la métrica de distancia** y del **valor de** k puede alterar la clase prevista para una misma muestra, incluso manteniendo fijo el descriptor utilizado — reforzando que, en el reconocimiento de patrones clásico, el descriptor, la métrica y el clasificador forman un sistema interdependiente, y no piezas aisladas.

7.18.6.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: entero C seguido de C nombres de clase.
- Línea 2: entero H, *string* M y entero k.
- Línea 3: entero N.
- Siguientes N líneas de entrenamiento: nombre de la clase seguido de H reales.
- Siguiente línea: entero Q.
- Siguientes Q líneas de prueba: nombre de la clase real seguido de H reales.

Salida:

- Q líneas con la clase prevista de cada muestra de prueba, en el orden de entrada.
- C líneas con la matriz de confusión (una fila por clase real).
- Última línea: `Acuracia: <valor>`.

7.18.6.5 📌 Ejemplos

Entrada (resumida)	Salida	Observación
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	Con $k = 1$, cada prueba se clasifica por el vecino de entrenamiento más cercano.

> **Nota**
>
> Este simulador usa un conjunto simplificado de **3 clases** (`granular`, `listrada`, `manchada`) sobre puntos 2D fictícios, solo para ilustrar el *pipeline* de votación, desempate y matriz de confusión del k-NN. En el **EP07_07**, aplicarás esta misma lógica a un mosaico de imagen real, que introduce una cuarta clase (`xadrez`) y sustituye los puntos 2D por histogramas LBP extraídos directamente de los píxeles de la imagen.

```
%%writefile EP07_06.py
# Código Python
```

```
Overwriting EP07_06.py
```

```
TestSuite("EP07_06.py").run()
```

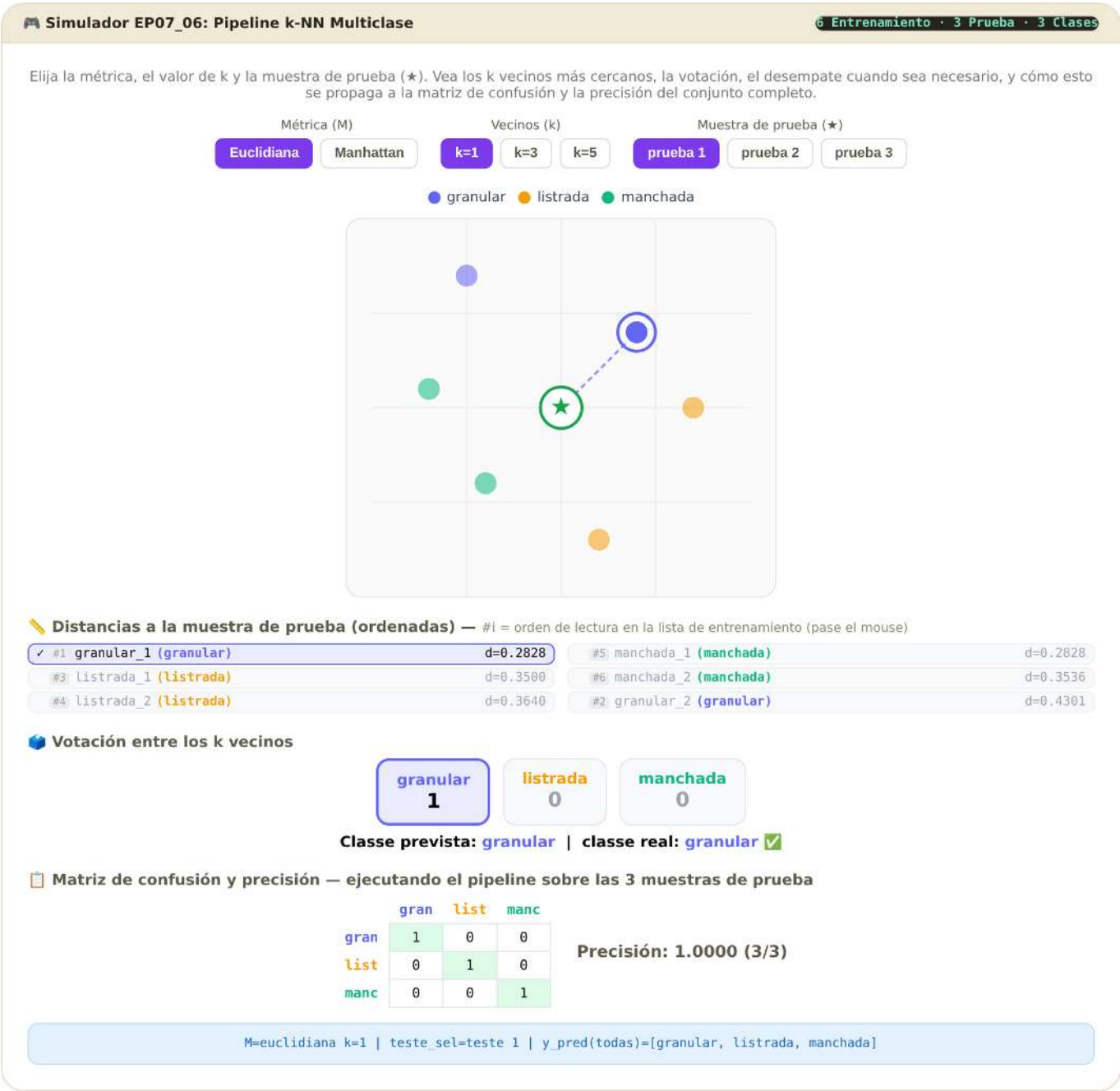
```
<IPython.core.display.HTML object>
```

7.18.7 EP07_07 ⚫ Clasificación Real de un Mosaico de Texturas mediante LBP + k-NN

En los ejercicios anteriores, el descriptor LBP (**EP07_04**) y el clasificador k-NN multiclase (**EP07_06**) se estudiaron por separado, siempre a partir de datos ya proporcionados en la entrada: vecindades 3×3 aisladas o histogramas previamente extraídos. En este ejercicio de cierre del capítulo, el programa deberá **leer una imagen real**, en formato **PGM ASCII (P2)**, calcular el descriptor LBP directamente a partir de los píxeles y, a continuación, clasificar cada región mediante el k-NN, reproduciendo, a escala reducida, el flujo completo de un sistema de reconocimiento de texturas. Este enfoque también anticipa la idea de **clasificación por mosaico de regiones**, relacionada con la segmentación semántica estudiada en un capítulo posterior.

El simulador interactivo del **EP07_06** utilizaba solo tres clases (`granular`, `listrada` y `manchada`) representadas por puntos bidimensionales ficticios. En este ejercicio, se añade una cuarta clase, **xadrez**, y los puntos se sustituyen por histogramas LBP extraídos de una imagen real.

La imagen de entrada es un **mosaico** formado por una cuadrícula $G \times G$ de bloques cuadrados de $S \times S$ píxeles. Cada bloque contiene una muestra de una de las cuatro clases de textura sintética del capítulo: **granular**, **listrada**, **manchada** o **xadrez** (patrón de tablero con intensidades alternadas). Como en los demás ejercicios del libro, la carga de la imagen se realiza mediante la función didáctica `mm.readImg`.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap07/sim-ep0706.html

Figura 7.26: Simulador EP07_06: *Pipeline* k-NN Multi-Clase (votación, desempate y matriz de confusión)

¿Por qué un mosaico único y no varias imágenes?

La entrada reúne las $G \times G$ muestras de textura en un único archivo **PGM**, solo para simplificar la lectura de los datos y evitar la apertura de varios archivos. Para el algoritmo, esto no altera el procesamiento: cada bloque se trata de forma independiente, como si fuera una imagen aislada. La única excepción es la **exclusión del borde** (ítem 4 a continuación).

7.18.7.1 📋 Directrices de Implementación

1. **Lectura de las dimensiones de la imagen**

   Leer, mediante la entrada estándar, dos líneas que contengan, respectivamente, el número de filas L y el número de columnas C del mosaico (ambos múltiplos del tamaño de bloque S, con $L = C$).

2. **Carga de la imagen**

   Utilizar la función didáctica

   ```
   f = mm.readImg(L, C)
   ```

   para leer los $L \times C$ valores de intensidad (tonos de gris, `uint8`) del mosaico.

3. **Parámetros de la cuadrícula**

   Leer el entero G (número de bloques por lado) y el entero S (tamaño del lado de cada bloque, en píxeles), satisfaciendo $L = C = G \times S$.

4. **Cálculo del código LBP por píxel**

   Para cada píxel **interior** de la imagen (es decir, que no esté en el borde global de `f` — fila o columna 0 o $L-1/C-1$), calcule el código LBP con $P = 8$ vecinos y radio $R = 1$, recorriendo los vecinos en sentido **horario** a partir de la esquina superior izquierda, exactamente como en el 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]`.

   Los píxeles en el borde global de la imagen **no** poseen vecindad completa y deben ser **ignorados** (no contribuyen a ningún histograma). Esto incluye píxeles de borde que caen en el interior de un bloque (la exclusión es siempre respecto al borde de la imagen completa, no al borde de cada bloque).

5. **Histograma LBP uniforme por bloque (10 compartimentos)**

   Para cada bloque (i, j) de la cuadrícula $(i, j = 0, \dots, G-1)$, acumule, entre sus píxeles válidos (ítem 4), un histograma $H^{(i,j)}$ de 10 compartimentos:

   - Considerando la secuencia circular de bits $s_0, \dots, s_7$ del píxel (misma regla de transiciones del EP07_04): si el número de transiciones es ≤ 2 (patrón **uniforme**), el píxel contribuye al compartimento $\text{popcount}(s_0, \dots, s_7) \in \{0, \dots, 8\}$ (número de bits iguales a `1`);
   - En caso contrario (patrón **no uniforme**), el píxel contribuye al compartimento 9.

   Al final, normalice el histograma de cada bloque dividiendo por el número de píxeles válidos contenidos en él, obteniendo $\hat{H}^{(i,j)}$, con $\sum_{b=0}^{9} \hat{H}^{(i,j)}[b] = 1$.

6. **Prototipos de entrenamiento**

   Leer el entero Ncl (número de clases) seguido de Ncl nombres de clase (orden que define la matriz de confusión y el desempate de votación, como en el EP07_06); a continuación, leer la *cadena* M (métrica: `euclidiana` o `manhattan`) y el entero impar k; por último, leer el entero N (número de prototipos) y, para cada uno, el nombre de la clase seguido de 10 valores reales (histograma prototipo ya normalizado).

7. **Clasificación k-NN de cada bloque**

Para cada bloque, calcule la distancia de $\widehat{H}^{(i,j)}$ a cada uno de los N prototipos, usando la métrica M (mismas fórmulas del EP07_06). Seleccione los k prototipos más cercanos (desempate de distancia por el orden de lectura de los prototipos) y clasifique por la clase mayoritaria (desempate de votación por el orden de las clases del ítem 6).

8. **Etiquetas reales y evaluación**

   Leer, en una única línea, los $G \times G$ nombres de clase **reales** de cada bloque, en orden de lectura por fila de la cuadrícula (bloque $(0,0)$, $(0,1)$, ..., $(0,G-1)$, $(1,0)$, ...). Construya la matriz de confusión $Ncl \times Ncl$ (fila = clase real, columna = clase predicha) y la precisión global.

9. **Salida**

   Imprimir, para cada bloque (en el mismo orden de lectura de las etiquetas reales del ítem 8), la clase predicha. A continuación, imprimir la matriz de confusión (una fila por clase real, en el orden del ítem 6). Por último, imprimir la precisión, redondeada a 4 decimales.

7.18.7.2 📌 Restricciones Computacionales

- **Descriptor fijo:** $P = 8$, $R = 1$ y 10 compartimentos (según el ítem 5) son fijos en este ejercicio — no se leen de la entrada.
- **Exclusión de borde global, no de bloque:** un píxel en el límite entre dos bloques, pero en el interior de la imagen, es válido y contribuye normalmente al histograma del bloque al que pertenece.
- **Orden de lectura como criterio de desempate:** tanto el desempate de distancia (ítem 7) como el de votación (ítem 7) siguen exactamente las mismas convenciones del EP07_01 y del EP07_06.
- **Prototipos como entrada, no aprendidos:** a diferencia del Proyecto Práctico 2, los histogramas de entrenamiento se proporcionan directamente en la entrada; el programa no debe generar texturas sintéticas.

7.18.7.3 🧠 Fundamentación Teórica

Etapa del ejercicio	Etapa correspondiente en el capítulo
Lectura de la imagen mediante `mm.readImg`	Adquisición de la imagen en el *pipeline* de reconocimiento de patrones
Código LBP por píxel (EP07_04)	`local_binary_pattern(imagen, P=8, R=1, method="uniform")`
Histograma de 10 compartimentos por bloque	Función `descritor_lbp` del Proyecto Práctico 2 (`bins=10`, `range=(0, P+2)`)
Clasificación k-NN con métrica seleccionable (EP07_06)	`KNeighborsClassifier` entrenado sobre `X_textura`
Matriz de confusión $Ncl \times Ncl$ y precisión	`confusion_matrix` y `accuracy_score` sobre `yt_teste`

Este ejercicio evidencia, con píxeles reales en lugar de valores sintéticos, una limitación discutida en la sección final del capítulo: clases de textura visualmente distintas para un observador humano — como **granular** y **manchada** — pueden producir histogramas LBP similares cuando la vecindad considerada es pequeña ($R = 1$), pues ambas presentan alta frecuencia de patrones no uniformes en la escala de un único píxel. En cambio, la clase **xadrez**, por poseer bordes regulares y repetitivos, tiende a separarse con mayor facilidad. Se espera que la matriz de confusión producida refleje exactamente ese patrón de confusión parcial.

7.18.7.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

```
L
C
[matriz L x C de la imagen]
G S
Ncl nombre_clase_1 ... nombre_clase_Ncl
M k
N
nombre_clase h0 h1 ... h9      (repetida N veces)
etiqueta(0,0) etiqueta(0,1) ... etiqueta(G-1,G-1)
```

Salida:

- $G \times G$ líneas con la clase predicha de cada bloque, en el orden de lectura de la cuadrícula.
- Ncl líneas con la matriz de confusión (una fila por clase real, valores separados por espacio).
- Última línea: `Acuracia: <valor>`.

7.18.7.5 📌 Ejemplo (verificación manual)

Para comprobar la implementación del descriptor antes de probarla sobre un mosaico completo, considere una imagen 6×6 **homogénea**, con todos los píxeles de intensidad 100, tratada como un único bloque ($G = 1$, $S = 6$). Como todo píxel interior tiene los 8 vecinos con intensidad igual a la del centro ($g_p \geq g_c$ en todos los casos), todos los bits s_p valen `1`, el número de transiciones es 0 (uniforme) y el compartimento es popcount(11111111) = 8. El histograma del único bloque es, por tanto, `0 0 0 0 0 0 0 0 1 0`.

Entrada (resumida)	Salida	Observación
6 6 [36 valores iguales a 100] 1 6 2 uniforme outra euclidiana 1 2 uniforme 0 0 0 0 0 0 0 0 1 0 outra 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 uniforme	uniforme 1 0 0 0 Acuracia: 1.0000	Distancia del bloque al prototipo `uniforme` es exactamente 0; la clase `outra` no aparece en la etiqueta real, por eso su fila en la matriz de confusión es nula.

7.18.7.6 📌 Archivos de Referencia (.pgm)

Para depuración local, dos mosaicos de prueba en el patrón ASCII P2 están disponibles (anexados a esta entrega; al integrarlos al repositorio del capítulo, guárdelos en `all/cap07/dados/EP07/`):

- 📥 **Caso 1 — Mosaico simple (`Caso1_Mosaico_Simples.pgm`)**: cuadrícula 2×2 de bloques de 24×24 píxeles, una muestra de cada una de las cuatro clases, con bajo ruido — útil para validar la lectura de la imagen y la lógica de clasificación en un escenario controlado.
- 📥 **Caso 2 — Mosaico mixto (`Caso2_Mosaico_Misto.pgm`)**: cuadrícula 3×3 de bloques de 16×16 píxeles, con clases repetidas y mayor variabilidad — escenario en el que la confusión entre **granular** y **manchada** discutida en la Fundamentación Teórica tiende a manifestarse.

La Figura 7.27 exhibe los dos mosaicos, para inspección visual antes de la implementación.

```python
import os
import urllib.request
import numpy as np

def garantir_e_baixar_arquivo(nome_arquivo):
    diretorio_local = "dados/EP07"
```

```
    caminho_local = os.path.join(diretorio_local, nome_arquivo)

    # Crear el directorio local si no existe
    if not os.path.exists(diretorio_local):
        os.makedirs(diretorio_local)

    # Si el archivo no existe localmente, se descarga del repositorio remoto
    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"Descargando {nome_arquivo} desde 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)

# Garantiza la descarga y obtiene la ruta correcta
arq_caso1 = garantir_e_baixar_arquivo("Caso1_Mosaico_Simples.pgm")
arq_caso2 = garantir_e_baixar_arquivo("Caso2_Mosaico_Misto.pgm")

# Lee las matrices PGM
caso1 = ler_pgm_p2(arq_caso1)
caso2 = ler_pgm_p2(arq_caso2)

mm.show(
    [caso1, caso2],
    titles=[
        "Caso 1: Mosaico Sencillo\n(2x2 bloques, 1 muestra/clase)",
        "Caso 2: Mosaico Mixto\n(3x3 bloques, clases repetidas)",
    ],
    cols=2,
    figsize=(8, 4),
)
```

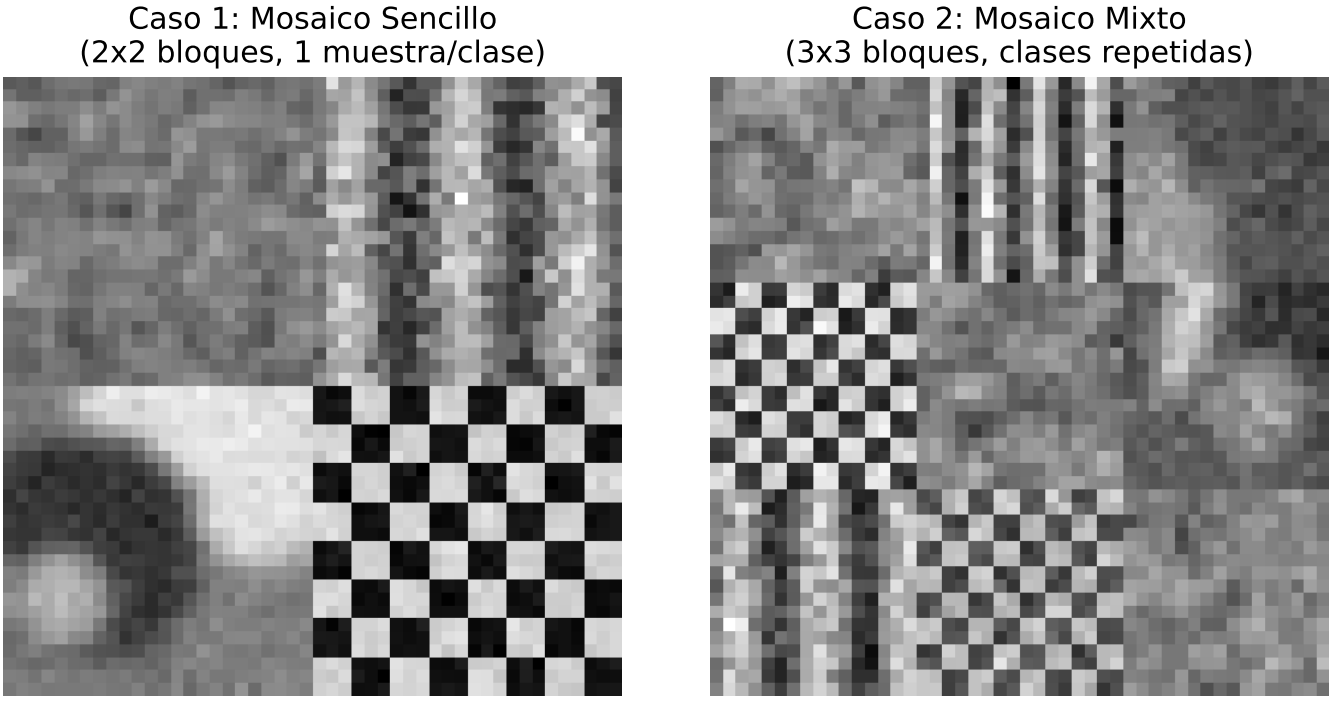
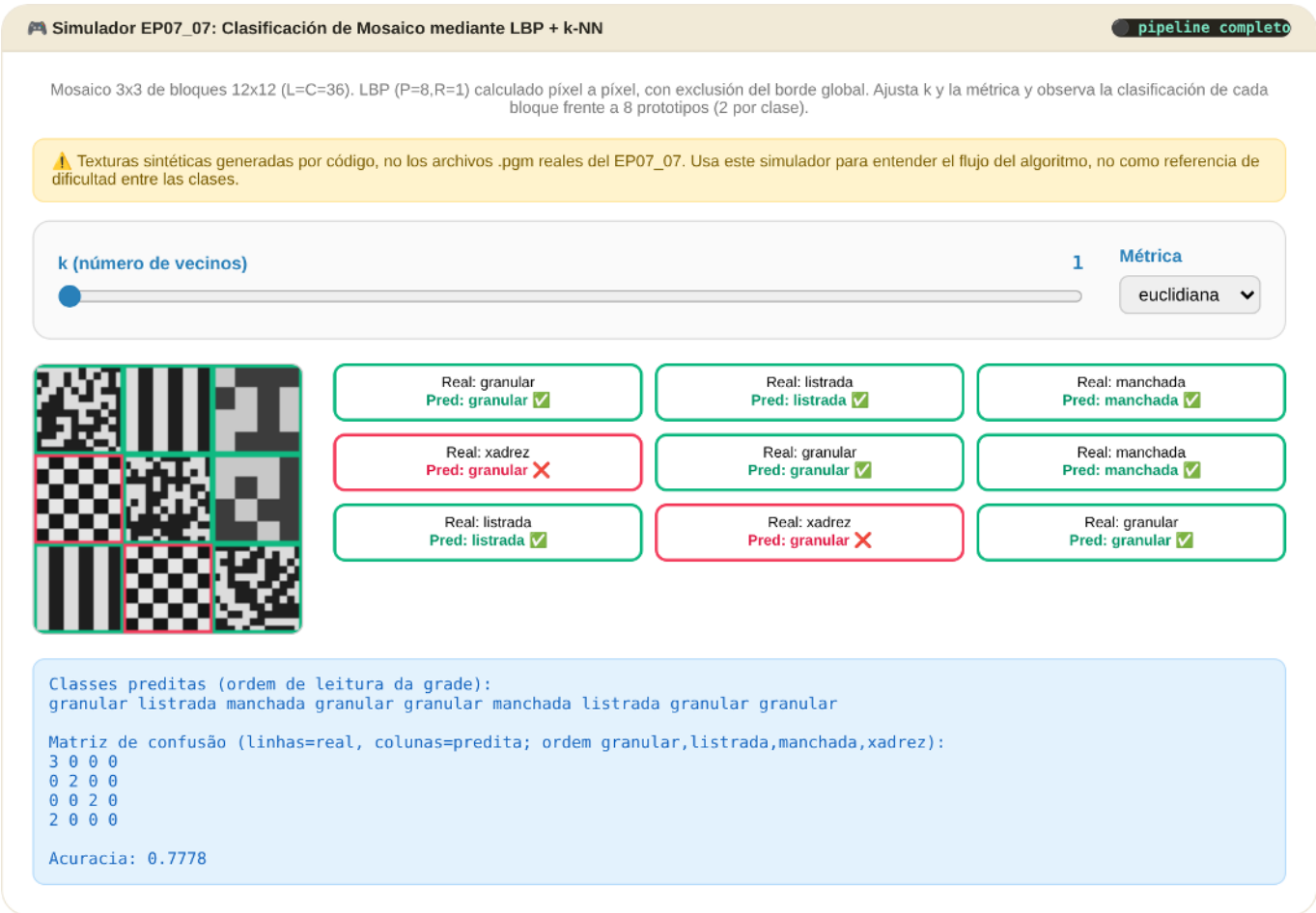


Figura 7.27: Mosaicos de referencia (formato PGM ASCII) utilizados en el EP07_07. Caso 1: cuadrícula 2x2 con una muestra de cada clase. Caso 2: cuadrícula 3x3 con clases repetidas y mayor variabilidad.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap07/sim-ep0707.html

Figura 7.28: Simulador EP07_07: Clasificación de un Mosaico de Texturas vía LBP + k-NN

```
%%writefile EP07_07.py
# Código Python
```

```
Overwriting EP07_07.py
```

```
TestSuite("EP07_07.py").run()
```

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

Capítulo 8

Comprendiendo Escenas: Correspondencia de Características, Detección y Segmentación

Executar Colab Abrir GitHub

En el **Capítulo 7**, se estudió cómo representar imágenes mediante descriptores y utilizar esas representaciones para tareas de clasificación. En este capítulo, el problema se amplía: además de reconocer patrones, se hace necesario establecer correspondencias entre diferentes imágenes, localizar automáticamente objetos de interés e interpretar la organización espacial de una escena.

Estos problemas constituyen algunas de las principales tareas de la Visión Computacional y representan una etapa natural después de la clasificación de imágenes. Para resolverlos, se presentarán métodos clásicos de correspondencia de características, detección de objetos y segmentación de imágenes, así como una visión general de los enfoques modernos basados en *Deep Learning*, preparando la transición hacia el **Capítulo 9**.

8.1 Objetivos del Capítulo

Al concluir este capítulo, el estudiante deberá ser capaz de:

- **Establecer correspondencias entre imágenes** utilizando detectores y descriptores locales, estimando transformaciones geométricas mediante homografías para el registro y la corrección de perspectiva;
- **Localizar objetos de interés** en imágenes utilizando métodos clásicos de detección y evaluar los resultados mediante métricas como la ***Intersection over Union*** **(IoU)** y la técnica de ***Non-Maximum Suppression*** **(NMS)**;
- **Diferenciar los paradigmas de segmentación** —semántica, de instancias y panóptica— comprendiendo que la segmentación panóptica unifica la segmentación semántica y la segmentación de instancias, proporcionando una descripción más completa de la escena, y aplicar métodos clásicos de segmentación;
- **Extraer descriptores geométricos y topológicos** de objetos segmentados y exportarlos como anotaciones estructuradas (CSV o formato YOLO), validando la calidad de las anotaciones mediante la métrica IoU;
- **Relacionar los métodos clásicos de correspondencia, detección y segmentación** con los enfoques modernos basados en *Deep Learning*, estudiados en el **Capítulo 9**.

La Figura 8.1 presenta una visión general de los principales conceptos y de las relaciones entre los temas abordados en este capítulo, sirviendo como un mapa conceptual para orientar la lectura.

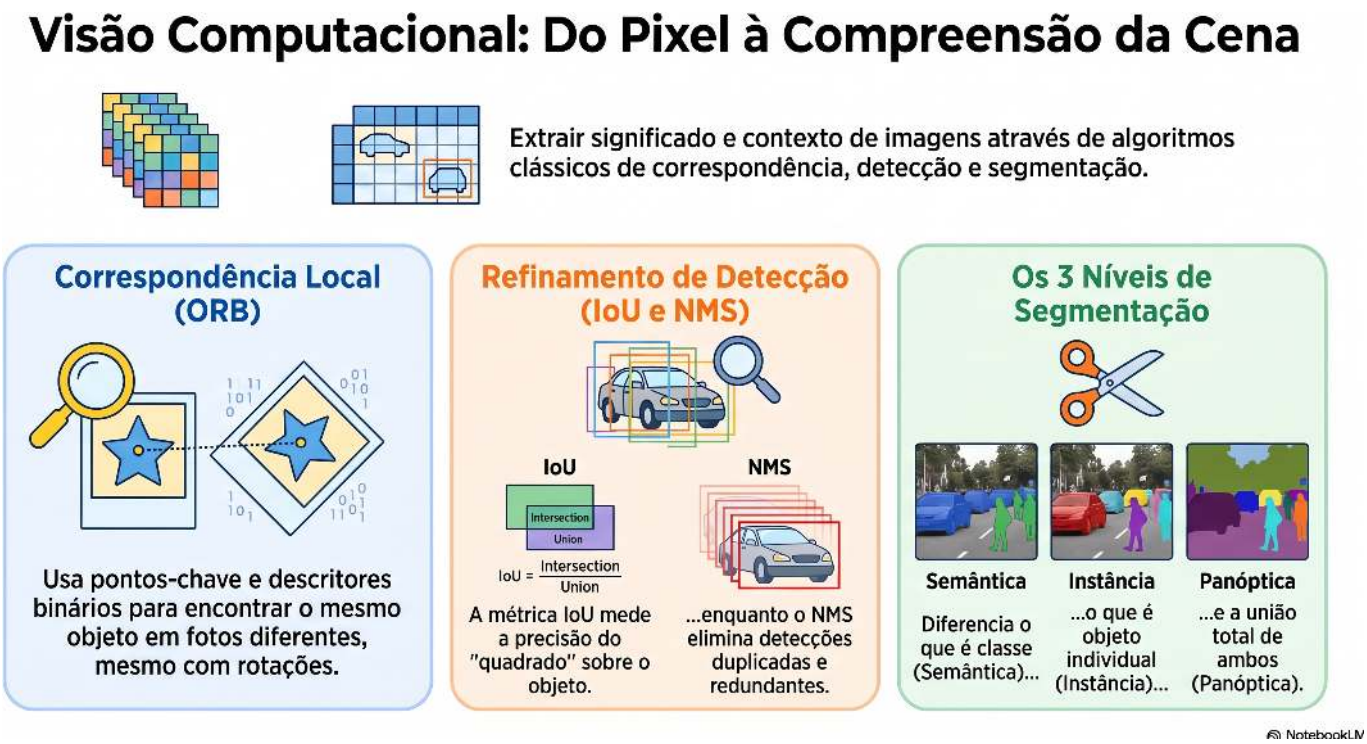


Figura 8.1: Visión general de los principales conceptos abordados en este capítulo, incluyendo correspondencia de características, detección de objetos, segmentación de imágenes y sus relaciones con enfoques modernos basados en *Deep Learning*. **Fuente:** elaborado con ayuda de Gemini Notebook ({GOOGLE}, 2025).

8.2 Configuración del Entorno

```
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


import importlib
import subprocess
import sys

def setup_cap08():
    """Instala bibliotecas faltantes necesarias para este capítulo."""
    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"[AVISO] Fallo al instalar {pkg} (necesario para el módulo {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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

8.3 Detectores y Descriptores Locales de Características

En el capítulo anterior, cada imagen fue representada por un único vector de características utilizado para clasificación. En diversas aplicaciones, sin embargo, es necesario comparar solo partes de la imagen, estableciendo correspondencias entre regiones observadas en diferentes instantes, posiciones o puntos de vista. Esta tarea requiere una representación local de la imagen, capaz de identificar estructuras suficientemente distintas para ser reencontradas en otras imágenes.

Este proceso se realiza en dos etapas complementarias. Inicialmente, un **detector** identifica puntos de interés (*keypoints*), normalmente asociados a esquinas o regiones con variaciones significativas de intensidad. A continuación, un **descriptor** representa numéricamente la vecindad de cada punto detectado, permitiendo comparar regiones correspondientes entre diferentes imágenes.

Una vez obtenidos los pares *(punto, descriptor)*, la correspondencia (*matching*) consiste en encontrar, para cada descriptor de una imagen, el descriptor más semejante en la otra. Estas correspondencias constituyen la base de diversas aplicaciones, como registro de imágenes, reconstrucción tridimensional, navegación visual y realidad aumentada.

8.3.1 El Algoritmo ORB (*Oriented FAST and Rotated BRIEF*)

En este capítulo se utilizará el **ORB**, un detector y descriptor local que combina eficiencia computacional y robustez ante rotaciones. El algoritmo reúne tres componentes principales:

- **FAST** (*Features from Accelerated Segment Test*), responsable de la detección de los puntos clave;
- **BRIEF** (*Binary Robust Independent Elementary Features*), responsable de la construcción del descriptor binario;
- un mecanismo de **estimación de la orientación de la vecindad**, que hace que el descriptor sea aproximadamente invariante a la rotación.

El detector **FAST** recorre todos los píxeles de la imagen. Para cada píxel candidato, analiza un círculo de **16 píxeles** a su alrededor. Si un conjunto de píxeles consecutivos presenta una intensidad significativamente mayor o menor que la intensidad del píxel central, ese píxel se considera un **punto clave** (*keypoint*). A continuación, los candidatos muy cercanos se filtran, conservando solo los más representativos.

Tras la detección de los puntos clave, el descriptor **BRIEF** se calcula en una vecindad alrededor de cada punto clave, y no solo sobre los 16 píxeles utilizados por el FAST. En esa región, un **patrón de muestreo**, formado por un conjunto fijo de pares de puntos (x, y) distribuidos en una ventana alrededor del punto clave, se utiliza para realizar comparaciones de intensidad según la Ecuación 8.1. En el ORB, la **orientación predominante de la vecindad** se estima a partir de la distribución de las intensidades en esa región, y el

patrón de muestreo se rota de acuerdo con dicha orientación. Además, el ORB utiliza una versión optimizada del BRIEF, denominada **rBRIEF** (*Rotated BRIEF*), en la cual los pares de puntos se seleccionan para producir descriptores más discriminativos y con baja correlación entre sus bits.

$$\tau(p; x, y) = \begin{cases} 1, & \text{si } I(x) < I(y), \\ 0, & \text{en caso contrario,} \end{cases} \tag{8.1}$$

donde:

- p es el punto clave detectado por el FAST;
- x e y son dos puntos de la vecindad de p, elegidos por el patrón de muestreo del BRIEF;
- $I(\cdot)$ representa la intensidad de un píxel;
- $\tau(p; x, y)$ es el resultado de la comparación binaria entre los puntos x e y.

Cada comparación genera un bit del descriptor. La concatenación de todas estas comparaciones forma el descriptor binario asociado al punto clave.

Como el descriptor es binario, la similitud entre dos puntos se mide mediante la **distancia de Hamming**, correspondiente al número de bits diferentes entre dos descriptores. Esta métrica puede calcularse de forma muy eficiente mediante operaciones lógicas sobre los bits, lo que hace que el ORB sea adecuado para aplicaciones en tiempo real.

8.3.2 Explorando el Simulador del ORB

La Figura 8.3 ilustra, de forma interactiva, cómo se construye el descriptor **ORB**. Cada segmento representa uno de los **32 pares de puntos** (x, y) utilizados en la Ecuación 8.1.. El color del segmento indica el resultado de la comparación de intensidades: **verde** cuando $I(x) < I(y)$ (bit igual a 1) y **rojo** en caso contrario (bit igual a 0). La flecha **amarilla** representa la orientación predominante de la vecindad, estimada a partir del centroide de intensidad. El patrón de muestreo del BRIEF se rota de acuerdo con esta orientación, haciendo que el descriptor sea aproximadamente invariante a la rotación.

Inicialmente, deje el ruido en cero y compare la secuencia de bits con el *slider* en 0° y luego en 15° (anote el valor del campo "Firma del Descriptor Binario" en cada caso):

```
0°  : 00000000000000100000011111100010
15° : 00000000000000100000011111100010
```

En este ejemplo, ambos descriptores son idénticos, y el panel "Dist. Hamming" confirma una distancia de cero — evidencia de que la estimación de orientación está compensando correctamente la rotación de la imagen.

Ahora haga clic en "Agregar Ruido" y repita el experimento. Dado que el ruido se genera aleatoriamente en cada ejecución, los valores a continuación son solo un ejemplo — los suyos serán diferentes, pero deben presentar una distancia de Hamming de magnitud similar (típicamente entre 6 y 14 bits, de un total de 32):

```
0°  : 01000000001000110110011111101010
15° : 10110000001011100100111111100010
```

El ruido altera parte de las comparaciones de intensidades, modificando algunos bits del descriptor. La diferencia entre dos descriptores se mide mediante la **distancia de Hamming**, que corresponde al número de posiciones en las que los bits difieren. Para los descriptores anteriores, esta distancia es igual a **10**.

Una forma eficiente de calcular esta distancia en Python consiste en aplicar la operación **XOR** (`^`), que identifica los bits diferentes, seguida del método `bit_count()`, que contabiliza cuántos bits iguales a 1 existen en el resultado.

```
def hamming(a: int, b: int) -> int:
    return (a ^ b).bit_count()
a = 0b01000000001000110110011111101010
b = 0b10110000001011100100111111100010
hamming(a, b)
```

```
10
```

La Figura 8.2 compara esta implementación con una versión basada en la comparación de caracteres y con la implementación optimizada de OpenCV (`cv2.NORM_HAMMING`).

```
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   1.804937      1.0
1  XOR+bit_count()   0.013438    134.3
2           OpenCV   0.218675      8.3
```

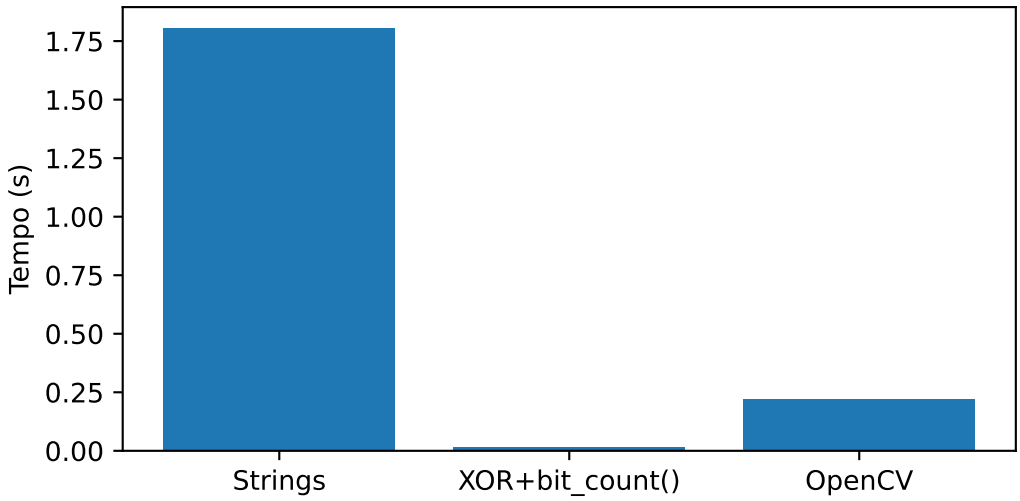
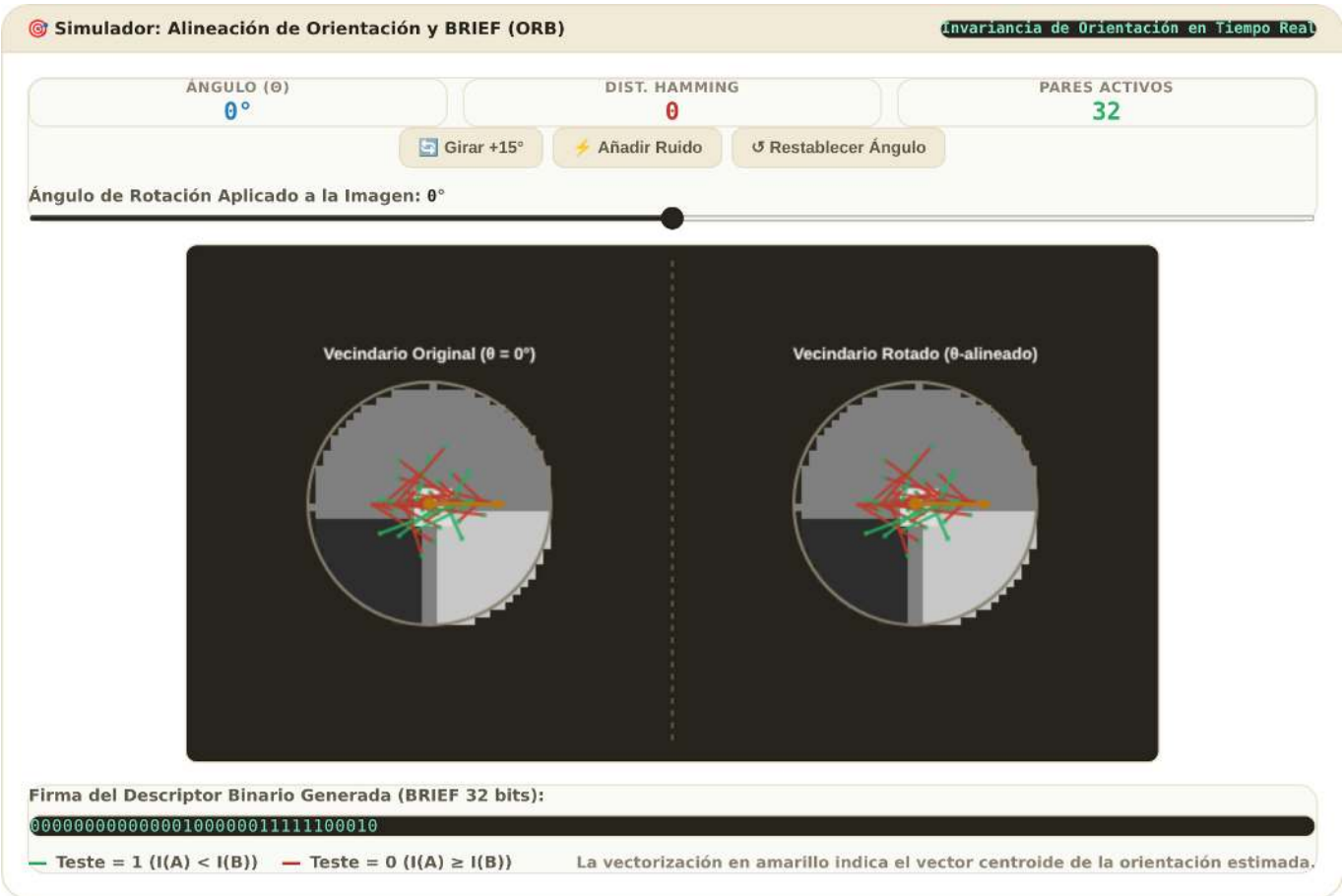


Figura 8.2: Comparación del desempeño de diferentes implementaciones de la distancia de Hamming.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap08/sim-08-orb.html

Figura 8.3: Simulador interactivo del descriptor ORB: explora la lógica de rotación y construcción del descriptor binario. Cambia la rotación para observar cómo el patrón de muestreo de pruebas binarias del BRIEF (líneas verdes y rojas) se orienta dinámicamente para garantizar la invariancia angular.

8.4 Correspondencia de Características y Homografía con ORB

Para ilustrar el proceso de correspondencia de características, se utiliza una escena sintética obtenida mediante la rotación de 20° de la imagen original. Esta transformación simula una segunda captura de la misma escena

bajo otro punto de vista. La Figura 8.4 presenta la imagen de referencia y su versión rotacionada, que se utilizarán en las etapas siguientes.

```
caminho_local = "dados/fig-08-Gemini-faces3.png"
url_imagem = "https://raw.githubusercontent.com/fzampirolli/pdi-vc/master/all/cap08/"
url_imagem += caminho_local

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", "Rotación (20°)"],
        cols=2, figsize=(8, 4))
```

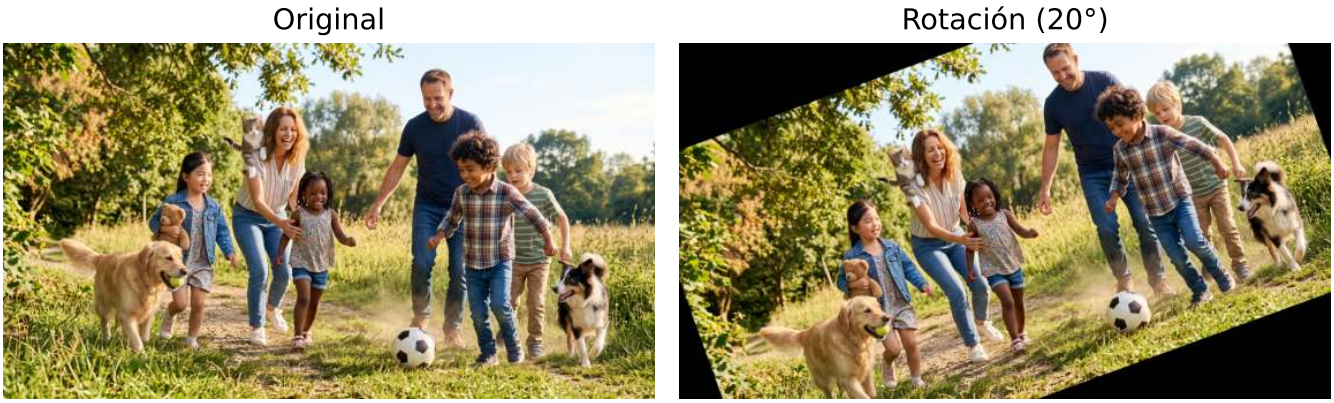


Figura 8.4: Imagen original generada por Gemini y versión rotacionada (20°), simulando un cambio de punto de vista.

8.4.1 Detectando y Correspondiendo Características con ORB

Con las dos imágenes disponibles, el **ORB** detecta los puntos clave y calcula sus descriptores binarios. A continuación, el **BFMatcher** de la librería `cv2` establece las correspondencias entre los descriptores utilizando la distancia de Hamming y la verificación mutua (*cross-check*). Finalmente, las correspondencias se ordenan de menor a mayor distancia de Hamming, priorizando los pares potencialmente más confiables. En el código siguiente, que genera la Figura 8.5, se destacan los siguientes pasos:

1. `cv2.ORB_create(nfeatures=500)` instancia el detector ORB, limitando la búsqueda a los **500 puntos clave** más representativos de cada imagen. Esta restricción reduce el costo computacional y evita la selección de puntos poco distintivos.
2. `orb.detectAndCompute(...)` ejecuta, en una sola llamada, la detección de los puntos clave mediante el **FAST** y el cálculo de los descriptores mediante el **BRIEF orientado**, devolviendo la lista de puntos clave (`kp`) y sus descriptores binarios de **256 bits** (`des`).
3. `cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)` crea un comparador por fuerza bruta (*Brute-Force Matcher*), que utiliza la **distancia de Hamming** —la misma métrica explorada en la Figura 8.3— para comparar cada descriptor de la imagen original con todos los descriptores de la imagen rotada. El parámetro `crossCheck=True` conserva únicamente los pares en los que la mejor correspondencia es recíproca, es decir, cuando el mejor correspondiente de A es B y, simultáneamente, el mejor correspondiente de B es A. Este criterio elimina gran parte de las correspondencias ambiguas.
4. `matches = sorted(...)` ordena las correspondencias de menor a mayor distancia de Hamming. Cuanto menor sea esta distancia, mayor será la similitud entre los descriptores y, en consecuencia, mayor será la probabilidad de que la correspondencia sea correcta.

La Figura 8.5 presenta solo las cinco correspondencias con menor distancia de Hamming. Aunque estos pares son los más prometedores, todavía no existe ninguna restricción geométrica entre los puntos correspondientes. Como consecuencia, algunos enlaces pueden representar falsas correspondencias (*false matches*), lo que justifica el uso del **RANSAC** en la etapa siguiente para identificar únicamente las correspondencias geométricamente consistentes.

```
orb = cv2.ORB_create(nfeatures=500)
kp1, des1 = orb.detectAndCompute(img_original, None)
kp2, des2 = orb.detectAndCompute(img_cena, None)
print(f"Puntos de interés detectados: {len(kp1)} (original), {len(kp2)} (escena)")

bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = sorted(
    bf.match(des1, des2),
    key=lambda m: m.distance
)
print(f"Correspondencias encontradas: {len(matches)}")

def draw_matches_destacado(img1, kp1, img2, kp2, matches, espessura=2, raio_ponto=4,
                           seed=42, cor_fixa=None):
    """Dibuja las imágenes lado a lado con líneas conectando los puntos correspondientes.
    Si cor_fija=None, cada correspondencia recibe un color aleatorio (facilita distinguir
    enlaces individuales). Si cor_fija está definida (ej: verde), todas las líneas usan el
    mismo color - útil para resaltar un subconjunto específico, como los inliers del RANSAC.
    """
    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)  # semilla fija = colores reproducibles en cada
ejecución

    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

# prueba las 5 peores: 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 Correspondencias ORB"], cols=1, figsize=(10, 5))
```

```
Puntos de interés detectados: 500 (original), 500 (escena)
Correspondencias encontradas: 300
```

Figura 8.5: Las 5 mejores correspondencias de características ORB entre la imagen original y la escena sintética, antes del filtrado por RANSAC.

La función `draw_matches_destacado()` tiene únicamente fines de visualización: posiciona las imágenes lado a lado y dibuja líneas entre los pares correspondientes, sin interferir en la estimación de las correspondencias.

8.5 Modelagem Matemática: Homografía y RANSAC

8.5.1 Homografía

En los Ejercicios 10 y 11 del Capítulo 2, las funciones `cv2.getPerspectiveTransform` y `cv2.warpPerspective` se utilizaron para corregir la perspectiva de imágenes a partir de **cuatro pares de puntos correspondientes** informados manualmente. En este capítulo, estas correspondencias pasan a obtenerse automáticamente mediante ORB, lo que permite estimar la transformación entre dos imágenes sin intervención del usuario.

Matemáticamente, esta transformación se describe mediante una **homografía**, representada por una matriz 3×3 que relaciona las coordenadas de un mismo plano observado desde diferentes puntos de vista:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

Tras la normalización de las coordenadas homogéneas, se obtiene el punto correspondiente

$$\left(\frac{x'}{w'}, \frac{y'}{w'} \right).$$

Como la homografía se define salvo un factor de escala, posee **ocho grados de libertad**. En consecuencia, se necesitan, como mínimo, **cuatro pares de puntos correspondientes** para estimar sus parámetros.

En la práctica, sin embargo, las correspondencias producidas automáticamente por ORB pueden contener asociaciones incorrectas (*outliers*). Para estimar la homografía de manera confiable incluso en presencia de estos errores, se utiliza el algoritmo **RANSAC**, presentado en la siguiente sección.

8.5.2 RANSAC

El **RANSAC** (*Random Sample Consensus*) es un algoritmo de estimación robusta capaz de ajustar un modelo geométrico incluso en presencia de observaciones incorrectas (*outliers*). En este capítulo, el modelo de interés es una **homografía**, estimada a partir de las correspondencias producidas por el ORB.

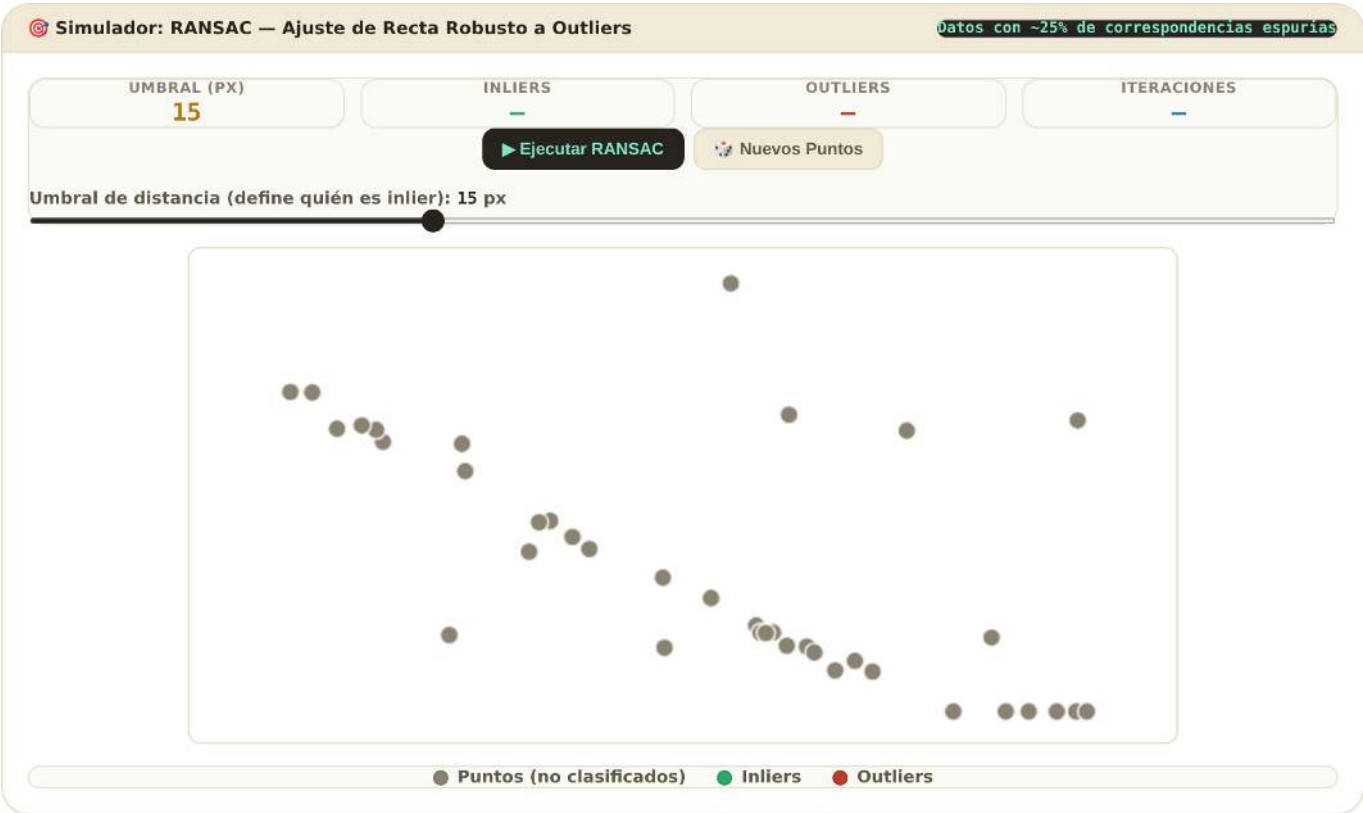
En cada iteración, el algoritmo:

1. selecciona aleatoriamente un pequeño subconjunto de correspondencias (cuatro pares de puntos, en el caso de la homografía);
2. estima una homografía candidata a partir de ese subconjunto;
3. verifica cuáles correspondencias son compatibles con esa transformación, clasificándolas como ***inliers*** o ***outliers***;
4. registra la homografía que produce el mayor número de ***inliers***;
5. reestima la homografía utilizando únicamente los ***inliers*** encontrados.

Aunque el ejemplo de este capítulo utilice una homografía, el RANSAC es un algoritmo de propósito general y puede emplearse para estimar diversos modelos geométricos, como rectas, circunferencias, planos y otras transformaciones. En todos los casos, el principio es el mismo: generar modelos candidatos a partir de pequeñas muestras aleatorias y seleccionar aquel que presente el mayor consenso entre los datos.

Para comprender este proceso de forma gradual, se presentan dos simuladores.

El primero, mostrado en la Figura 8.6, utiliza el ejemplo más simple posible: el ajuste de una **recta** a un conjunto de puntos que contiene aproximadamente **25% de *outliers***. El objetivo es comprender las etapas fundamentales del algoritmo — muestrear, estimar un modelo, identificar los ***inliers*** y repetir el proceso — sin la complejidad del registro entre imágenes.



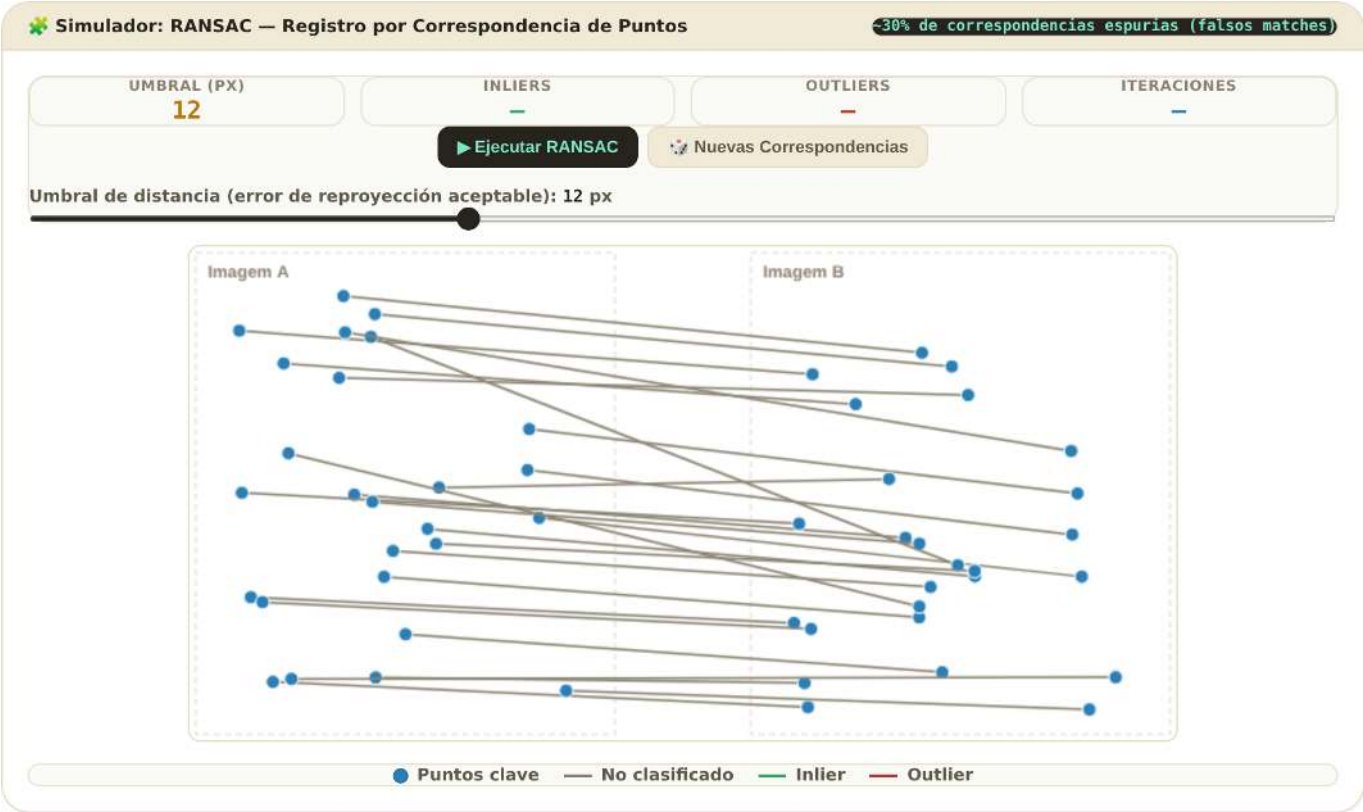
Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap08/sim-08-ransac.html

Figura 8.6: Simulador interactivo del algoritmo RANSAC: ajuste el umbral de distancia y ejecute el algoritmo para observar la separación entre *inliers* y *outliers*.

El segundo simulacro, presentado en la Figura 8.7, se aproxima al problema estudiado en este capítulo. En lugar de un único conjunto de puntos, se consideran **dos imágenes** que contienen correspondencias entre puntos clave. Algunas correspondencias son correctas (*inliers*), mientras que otras son incorrectas (*outliers*), resultantes de errores en el proceso de correspondencia de los descriptores. En este simulador, el modelo estimado es una **transformación de similitud** (rotación, escala y traslación), más simple que una homografía completa, pero suficiente para ilustrar el problema de registro entre imágenes.

En ambos simuladores, el algoritmo ejecutado sigue exactamente el mismo principio utilizado posteriormente para estimar la homografía. La única diferencia radica en el modelo geométrico ajustado.

Ajuste el umbral de distancia y ejecute el algoritmo en cada simulador para observar cómo el RANSAC identifica los ***inliers***, descarta los ***outliers*** y estima un modelo consistente utilizando únicamente las correspondencias válidas.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap08/sim-08-ransac-registro.html

Figura 8.7: Simulador interactivo de RANSAC aplicado al registro de imágenes: los puntos clave de dos imágenes se emparejan mediante un descriptor, algunas correspondencias son espurias (valores atípicos), y RANSAC estima la transformación de similitud que alinea la mayoría de ellas.

8.5.3 Estimando la Homografía con RANSAC

Tras obtener las correspondencias entre los puntos clave mediante ORB, el siguiente paso consiste en estimar la **homografía** entre las dos imágenes. Para ello, se utiliza la función `cv2.findHomography()`, que emplea el algoritmo **RANSAC** para calcular esta transformación y devolver una máscara que indica cuáles correspondencias fueron clasificadas como ***inliers***.

El siguiente código realiza cuatro operaciones principales:

1. extrae las coordenadas de los puntos correspondientes en cada imagen;
2. estima la homografía mediante `cv2.findHomography(..., cv2.RANSAC)`;
3. recibe la máscara producida por el RANSAC, en la cual cada correspondencia se clasifica como ***inlier*** o ***outlier***;
4. utiliza esta máscara para seleccionar únicamente las correspondencias clasificadas como ***inliers***.

La Figura 8.8 presenta solo las correspondencias clasificadas como ***inliers***. Se observa que estos pares de puntos son compatibles con una misma transformación geométrica, mientras que las correspondencias inconsistentes (*outliers*) se descartan. Como consecuencia, la homografía estimada representa de forma más fiel la relación geométrica entre las dos imágenes.

```
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"Matriz de homografía estimada:\n{H}\n")
print(f"Inliers: {n_inliers} de {len(matches)} correspondencias \
      ({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)  # verde (BGR)
)
mm.show([img_inliers], titles=[f"Correspondencias Inliers (RANSAC) - \
                               {n_inliers}/{len(matches)}"], cols=1, figsize=(10, 5))
```

```
Matriz de homografía estimada:
[[ 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 de 300 correspondencias       (95.0%)
```

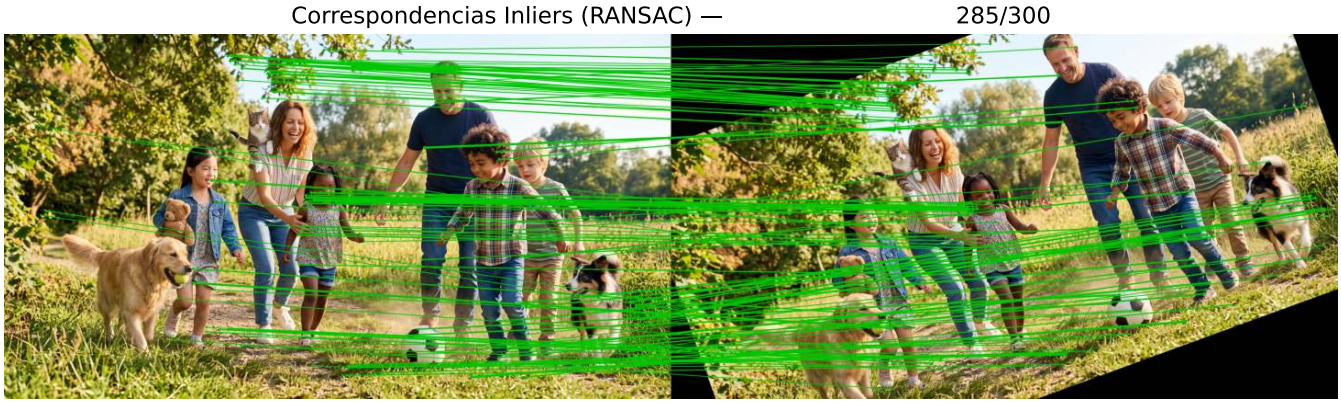


Figura 8.8: Correspondencias clasificadas como *inliers* (verde) por RANSAC al estimar la homografía entre las dos imágenes.

8.5.4 Registro de la Imagen

Tras estimar la homografía, el siguiente paso consiste en utilizarla para registrar la imagen de la escena en el sistema de coordenadas de la imagen original. Este proceso permite alinear las dos imágenes, facilitando la comparación entre ellas.

El código realiza tres operaciones principales:

1. aplica la transformación proyectiva mediante `cv2.warpPerspective()`, utilizando la opción `cv2.WARP_INVERSE_MAP`, que aplica internamente la transformación inversa sin necesidad de calcular explícitamente H^{-1};
2. calcula la diferencia absoluta píxel a píxel entre la imagen registrada y la imagen original mediante `cv2.absdiff()`;
3. muestra la imagen original, la escena rotada, la imagen registrada y un mapa de las diferencias entre las dos imágenes.

La Figura 8.9 presenta el resultado del registro. Se observa que la imagen registrada se vuelve visualmente muy similar a la imagen original, lo que indica que la homografía estimada logró alinear correctamente las dos vistas de la misma escena. El mapa de diferencias evidencia solo las regiones donde aún existen pequeñas discrepancias derivadas de errores de interpolación, cuantización y de la propia estimación de la homografía.

```
h, w = img_original.shape[:2]

# WARP_INVERSE_MAP: aplica H "de atrás hacia adelante", evitando el cálculo manual de 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( # muestra mm.gray(erro)>10 en niveles de gris
    [img_original, img_cena, img_registrada, mm.gray(erro)>10],
    titles=["Original", "Escena (rotada)", "Registrada", "Diferencia"],
    cols=2,
    figsize=(14, 8),
)
```

Figura 8.9: Registro de la imagen de la escena utilizando la homografía inversa estimada por RANSAC.

¿Por qué funciona? — Robustez por consenso

Muchos métodos de estimación ajustan un modelo utilizando **todas** las observaciones disponibles, buscando minimizar el error total entre los datos observados y el modelo ajustado (enfoque conocido como **mínimos cuadrados**). Cuando existen *outliers*, esas observaciones incorrectas pueden desplazar significativamente el resultado obtenido.

El **RANSAC** sigue una estrategia diferente. En lugar de utilizar todos los datos simultáneamente, estima modelos sucesivos a partir de pequeñas muestras aleatorias. Cada modelo se evalúa entonces por el número de correspondencias compatibles con él. Al final de las iteraciones, se selecciona el modelo que presenta el mayor consenso entre los datos, es decir, el mayor número de ***inliers***.

Dos parámetros desempeñan un papel fundamental en el algoritmo:

- el **umbral de reproyección**, que define la distancia máxima para que una correspondencia sea clasificada como ***inlier***;
- el **número de iteraciones**, que debe ser suficientemente grande para aumentar la probabilidad de seleccionar al menos una muestra libre de ***outliers***.

Aunque bastante robusto, el RANSAC presupone que existe un modelo geométrico predominante en los datos. Su desempeño tiende a disminuir cuando la proporción de ***inliers*** es muy pequeña o cuando diferentes estructuras geométricas coexisten en la misma escena, dificultando la identificación de un único modelo dominante.

8.6 Detección de Objetos: *Haar Cascade* (Viola-Jones)

La correspondencia de características responde a la pregunta: **"¿dónde está el mismo objeto o patrón observado anteriormente?"**. La **detección de objetos** resuelve un problema más general: localizar automáticamente instancias de una **categoría** (por ejemplo, rostros humanos), incluso si los objetos específicos nunca han sido observados durante el entrenamiento. Mientras que el ORB necesita dos imágenes para establecer correspondencias entre puntos, el **Haar Cascade** opera sobre una sola imagen, identificando directamente las regiones candidatas a contener el objeto buscado.

El algoritmo ***Haar Cascade***, propuesto por Viola y Jones [Viola; Jones (2001); Viola (2004)], combina características Haar, imágenes integrales y una cascada de clasificadores para realizar la detección de objetos de manera eficiente. Aunque actualmente existen métodos más recientes basados en redes neuronales convolucionales, el Haar Cascade permanece disponible en la biblioteca OpenCV y constituye un ejemplo clásico para el estudio de técnicas de detección de objetos.

Su funcionamiento se basa en tres componentes principales:

1. **Características Haar:** filtros rectangulares simples que miden diferencias de intensidad entre regiones vecinas de la imagen, explotando **patrones de contraste característicos** del objeto, como la región de los ojos generalmente más oscura que la frente;
2. **Imagen integral:** estructura de datos que permite calcular rápidamente la suma de los píxeles de cualquier región rectangular de la imagen:
$$I_{\text{integral}}(x, y) = \sum_{x' \leq x,\ y' \leq y} I(x', y'),$$
reduciendo significativamente el costo computacional de la evaluación de las características Haar. Con esta estructura, la suma de los píxeles de cualquier rectángulo puede obtenerse con solo cuatro accesos a la imagen integral;
3. **Cascada de clasificadores:** durante el entrenamiento, el algoritmo AdaBoost selecciona y combina clasificadores simples en una secuencia de etapas. En la detección, las regiones que claramente **no corresponden al objeto** se descartan desde las primeras etapas, mientras que solo las candidatas más prometedoras pasan por las etapas siguientes, más precisas y computacionalmente más costosas. Esta estrategia permite realizar una búsqueda eficiente en diferentes posiciones y escalas de la imagen.

El siguiente código, que genera la Figura 8.10, utiliza clasificadores entrenados previamente y disponibles en OpenCV para detectar **rostros** y, a continuación, restringe la búsqueda de **ojos** solo al interior de cada rostro detectado. Esta estrategia reduce falsos positivos y disminuye el costo computacional, ya que evita realizar la búsqueda de ojos en toda la imagen.

Durante la detección, una ventana recorre la imagen en diferentes posiciones y escalas. La función `detectMultiScale()` realiza esta búsqueda automáticamente. El parámetro `scaleFactor` controla el factor de reducción entre escalas consecutivas de la ventana, mientras que `minNeighbors` define el número mínimo de detecciones vecinas necesarias para confirmar un objeto, reduciendo detecciones espurias. El parámetro `minSize` establece el tamaño mínimo del objeto considerado durante la búsqueda.

La función devuelve una lista de rectángulos, cada uno descrito por las coordenadas de la esquina superior izquierda y las dimensiones `(x, y, ancho, alto)`. Estos rectángulos delimitan las regiones clasificadas como objetos por el detector y se utilizan para dibujar las cajas mostradas en la Figura 8.10.

```
def get_cascade(nome):
    """Descarga (si es necesario) y carga un clasificador Haar Cascade de OpenCV."""
```

```
    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)

# Más utilizado
#face_cascade = get_cascade("haarcascade_frontalface_default.xml")

# Más preciso, aunque más lento
face_cascade = get_cascade("haarcascade_frontalface_alt2.xml")

# Compromiso entre velocidad y precisión
#face_cascade = get_cascade("haarcascade_frontalface_alt.xml")

# Muy rápido, aunque menos preciso
#face_cascade = get_cascade("haarcascade_frontalface_alt_tree.xml")

# para los ojos, más utilizado
#eye_cascade = get_cascade("haarcascade_eye.xml")
eye_cascade  = get_cascade("haarcascade_eye_tree_eyeglasses.xml")

img_original = mm.read(caminho_local)

# Escala de grises + ecualización de histograma (Capítulo 3), como espera Haar Cascade
img_rgb = mm.rotate(img_original, angle=0)
img_gray = cv2.equalizeHist(cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY))

# Detecta caras y, dentro de cada una, intenta detectar los ojos
faces = face_cascade.detectMultiScale(
    img_gray,
    scaleFactor=1.1,      # Pasos más grandes entre escalas: más rápido, pero menos sensible
    minNeighbors=3,       # Nº mínimo de detecciones superpuestas para confirmar una cara
    minSize=(90, 90)      # Ignora regiones candidatas menores de 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"Regiones detectadas como cara: {len(faces)}")
mm.show([img_anotada], titles=
        ["Detección Haar Cascade: caras (verde) y ojos (rojo)"],
        cols=1, figsize=(8, 8))
```

```
Regiones detectadas como cara: 5
```

Figura 8.10: Detección de caras y ojos con *Haar Cascade*.

¿Por qué funciona? — Y por qué también falla

En la imagen utilizada en este ejemplo, el clasificador detecta la cara y los ojos, pero también puede marcar una segunda región sobre parte del fondo de la imagen como si fuera una cara. Este es un ejemplo de **falso positivo**: la distribución local de intensidades en esa región es suficientemente similar a los patrones aprendidos durante el entrenamiento como para que la cascada la clasifique incorrectamente como un rostro.

Este comportamiento evidencia una de las principales limitaciones del *Haar Cascade*. Como el método basa su decisión únicamente en características Haar, es decir, diferencias de intensidad entre regiones rectangulares, no representa explícitamente la forma ni el significado de los objetos presentes en la imagen. Así, texturas y patrones de contraste similares a los encontrados en caras pueden producir detecciones incorrectas. Además, su rendimiento tiende a disminuir ante grandes variaciones de pose, oclusiones, expresiones faciales y condiciones de iluminación diferentes de las presentes en los datos utilizados para el entrenamiento.

A pesar de estas limitaciones, el *Haar Cascade* sigue siendo útil en aplicaciones que priorizan un bajo costo computacional. En situaciones que requieren una mayor capacidad de generalización ante variaciones en la apariencia de los objetos, los métodos modernos basados en redes neuronales profundas suelen presentar un mejor rendimiento.

8.7 Detecção de Objetos: Caixas Delimitadoras, IoU e NMS

Aunque los métodos de detección de objetos utilizan estrategias bastante diferentes — desde *Haar Cascade* hasta los detectores modernos basados en redes neuronales profundas —, sus resultados se representan normalmente mediante **cajas delimitadoras** (*bounding boxes*), definidas por las coordenadas $(x_{min}, y_{min}, x_{max}, y_{max})$.

8.7.1 *Intersection over Union* (IoU)

La métrica ***Intersection over Union*** **(IoU)** cuantifica la superposición entre dos cajas delimitadoras, por ejemplo, la detección producida por un algoritmo y la anotación de referencia (*ground truth*):

$$\mathrm{IoU}(A, B) = \frac{\text{área}(A \cap B)}{\text{área}(A \cup B)} \in [0, 1].$$

Cuanto más cercano a 1, mayor es la concordancia entre las cajas; el valor 0 indica ausencia de superposición. En evaluaciones de detectores de objetos, es común considerar una detección correcta cuando IoU $\geq 0{,}5$, aunque aplicaciones específicas pueden adoptar umbrales diferentes.

La métrica IoU no se utiliza únicamente para evaluar detectores. También constituye el criterio empleado por el algoritmo de **Supresión de No-Máximos** para decidir cuándo dos cajas delimitadoras representan el mismo objeto y, por lo tanto, una de ellas debe ser eliminada.

8.7.2 Supresión de No-Máximos (NMS)

Durante la detección, es común que varias cajas delimitadoras estén asociadas al mismo objeto. La **Supresión de No-Máximos** (*Non-Maximum Suppression*, NMS) elimina esta redundancia en tres etapas:

1. ordena las detecciones por la puntuación de confianza, de mayor a menor;
2. mantiene la caja de mayor confianza y elimina aquellas cuyo IoU con ella supera un umbral determinado;
3. repite el proceso con las cajas restantes hasta que no existan superposiciones relevantes.

El ejemplo de la Figura 8.11 ilustra este procedimiento con dos objetos, cada uno representado inicialmente por varias cajas superpuestas. En este ejemplo, cada caja delimitadora recibe una **puntuación de confianza**, que representa el grado de confianza del detector de que esa región contiene el objeto buscado.

En lugar de implementar el algoritmo manualmente, se utiliza la función `cv2.dnn.NMSBoxes`, que implementa la Supresión de No-Máximos empleada en diversos detectores modernos. La función `desenhar_caixas` muestra, lado a lado, las cajas antes y después del NMS, evidenciando la reducción de cinco detecciones a solo dos, preservando únicamente la caja de mayor confianza para cada objeto.

```
def desenhar_caixas(caixas, pontuacoes, indices, cor, tamanho=(450, 450)):
    """Dibuja las cajas (y sus puntuaciones) indicadas en `indices` sobre un fondo blanco."""
    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 espera cajas en el formato (x, y, ancho, alto)
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,   # ningún recorte por confianza mínima aquí
    nms_threshold=0.5      # umbral de IoU para considerar dos cajas redundantes
).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"Antes del NMS ({len(caixas)} cajas)", \
            f"Después del NMS ({len(mantidas)} caja(s))"],
    cols=2, figsize=(9, 4.5)
)
```

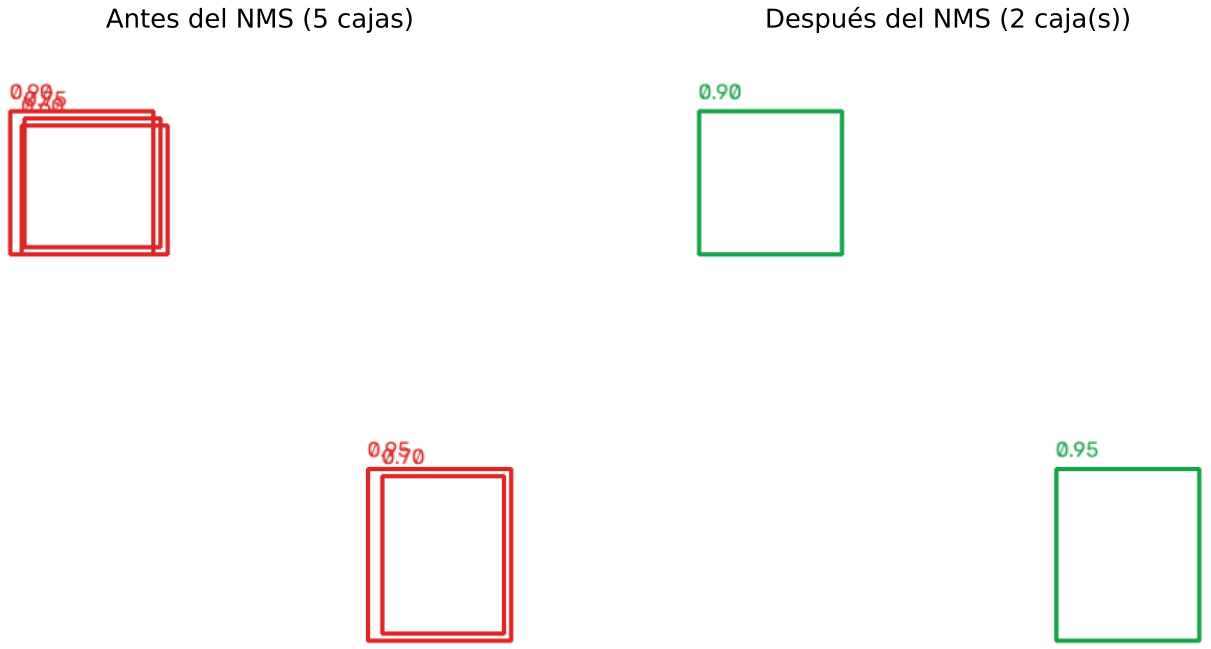


Figura 8.11: Efecto de la Supresión de No-Máximos (NMS): múltiples detecciones redundantes (izquierda) se reducen a una detección por objeto (derecha).

¿Por qué funciona? — Eliminación de redundancias

El **NMS** (*Non-Maximum Suppression*) no altera la calidad de las detecciones producidas por el detector. Su función es eliminar **cajas delimitadoras redundantes** que representan al mismo objeto, conservando únicamente aquella con mayor puntuación de confianza.
Su eficacia depende del umbral de IoU adoptado. Un valor muy bajo puede eliminar cajas correspondientes a objetos diferentes cuya superposición sea elevada, mientras que un valor muy alto puede mantener varias cajas superpuestas para un mismo objeto.
El NMS tampoco **elimina falsos positivos aislados**. Si una región incorrecta es detectada por una sola caja, esta se conservará, ya que no existe otra detección redundante con la cual compararla. Por este motivo, el NMS se aplica como una etapa de **posprocesamiento**, utilizando únicamente las cajas delimitadoras y sus puntuaciones de confianza producidas por el detector.

8.8 Segmentación Visual: Semántica, de Instancias y Panóptica

Una caja delimitadora indica aproximadamente la posición de un objeto, pero no identifica qué píxeles le pertenecen. La **segmentación visual** resuelve este problema asignando una etiqueta a cada píxel de la imagen. Según la información producida, se distinguen tres paradigmas principales, resumidos en la Tabla 8.1.

Tabla 8.1: Comparación entre los principales paradigmas de segmentación visual.

Paradigma	Pregunta que responde	¿Distingue objetos de la misma clase?
Semántica	"¿A qué clase pertenece cada píxel?"	No. Todos los píxeles de una misma clase reciben la misma etiqueta.
De instancias	"¿Qué píxeles pertenecen a cada objeto individual?"	Sí. Cada objeto recibe un identificador propio.
Panóptica	Combina las dos anteriores.	Sí. Cada píxel recibe una clase y, cuando corresponde, un identificador de instancia.

En la segmentación **semántica**, cada píxel recibe únicamente la etiqueta de su clase. En la segmentación **de instancias**, además de la clase, objetos distintos pertenecientes a la misma categoría se diferencian entre sí. Por su parte, la segmentación **panóptica** combina estas dos informaciones, asignando a cada píxel una clase y, cuando corresponde, un identificador de instancia.

8.8.1 Segmentación Semántica por Umbralización

Antes de la popularización de los métodos basados en redes neuronales profundas —tema del **Capítulo 9**—, muchas aplicaciones de segmentación se resolvían mediante técnicas clásicas de procesamiento de imágenes. Uno de los ejemplos más sencillos consiste en separar objeto y fondo mediante **umbralización**, produciendo una máscara binaria. Esta máscara caracteriza una **segmentación semántica**, pues cada píxel pasa a pertenecer a una de las dos clases: **moneda** o **fondo**.

La Figura 8.12 utiliza la imagen de monedas de la biblioteca `scikit-image`. Este ejemplo retoma los conceptos de umbralización y morfología matemática estudiados en el **Capítulo 4**, en sustitución del ejemplo anterior, que utilizaba otra imagen de monedas.

Inicialmente, se aplica un **cierre morfológico** para suavizar pequeñas imperfecciones en la superficie de las monedas. A continuación, la **umbralización de Otsu** produce la máscara binaria que separa monedas y fondo. Tras la eliminación de los objetos conectados al borde de la imagen, el relleno de regiones internas y la eliminación de pequeños ruidos mediante operaciones morfológicas, se obtiene una máscara semántica adecuada para las etapas siguientes.

8.8.2 Segmentación de Instancias por Etiquetado

Una máscara semántica informa únicamente la clase de cada píxel, pero no distingue objetos diferentes pertenecientes a la misma categoría. Para obtener una **segmentación de instancias**, se aplica el **etiquetado de componentes conectados**, que asigna un identificador diferente a cada región conexa de la máscara binaria.

En este ejemplo, cada componente conectado corresponde a una moneda individual. Así, el etiquetado produce una segmentación de instancias, permitiendo identificar, contar y medir por separado cada moneda presente en la imagen.

```
img_moedas = skdata.coins()
```

```python
# 1. Cierre morfológico: suaviza bordes y rellena pequeñas fallas en la
#    superficie de las monedas antes de la umbralización
fechamento = mm.close(img_moedas, mm.sedisk(4))

# 2. Umbralización de Otsu (Capítulo 4): separa monedas (claras) del fondo (oscuro)
mascara_bruta = mm.threshold(fechamento)

# 3. Elimina objetos conectados al borde de la imagen - monedas cortadas en las
#    extremidades no forman instancias completas y dificultarían el conteo
mascara_sem_borda = mm.edgeoff(mascara_bruta)

# 4. Cierre de agujeros: rellena posibles regiones internas no detectadas
#    por el umbral, garantizando que cada moneda sea un disco sólido
mascara_semantica = mm.clohole(mascara_sem_borda)

# 5. Apertura morfológica: elimina ruido residual y desconecta monedas que
#    por casualidad se tocan, preparando la máscara para el Etiquetado
mascara_limpa = mm.open(mascara_semantica, mm.sedisk(6))

# 6. Etiquetado de componentes conectados: cada moneda aislada recibe un
#    identificador de instancia distinto
rotulos_instancias = mm.label(mascara_limpa)

print(f"Instancias (monedas individuales) identificadas: {np.max(rotulos_instancias)}")

mm.show(
    [img_moedas, mascara_bruta, mascara_sem_borda, mascara_semantica,
     mascara_limpa, rotulos_instancias],
    titles=[
        "Original",
        "1. Umbralización (Otsu)",
        "2. Eliminación de bordes",
        "3. Relleno de agujeros\n(máscara semántica: moneda x fondo)",
        "4. Apertura (limpieza)",
        "5. Etiquetado\n(segmentación de instancias)"
    ],
    cols=3, figsize=(10, 6)
)
```

```
Instancias (monedas individuales) identificadas: 24
```

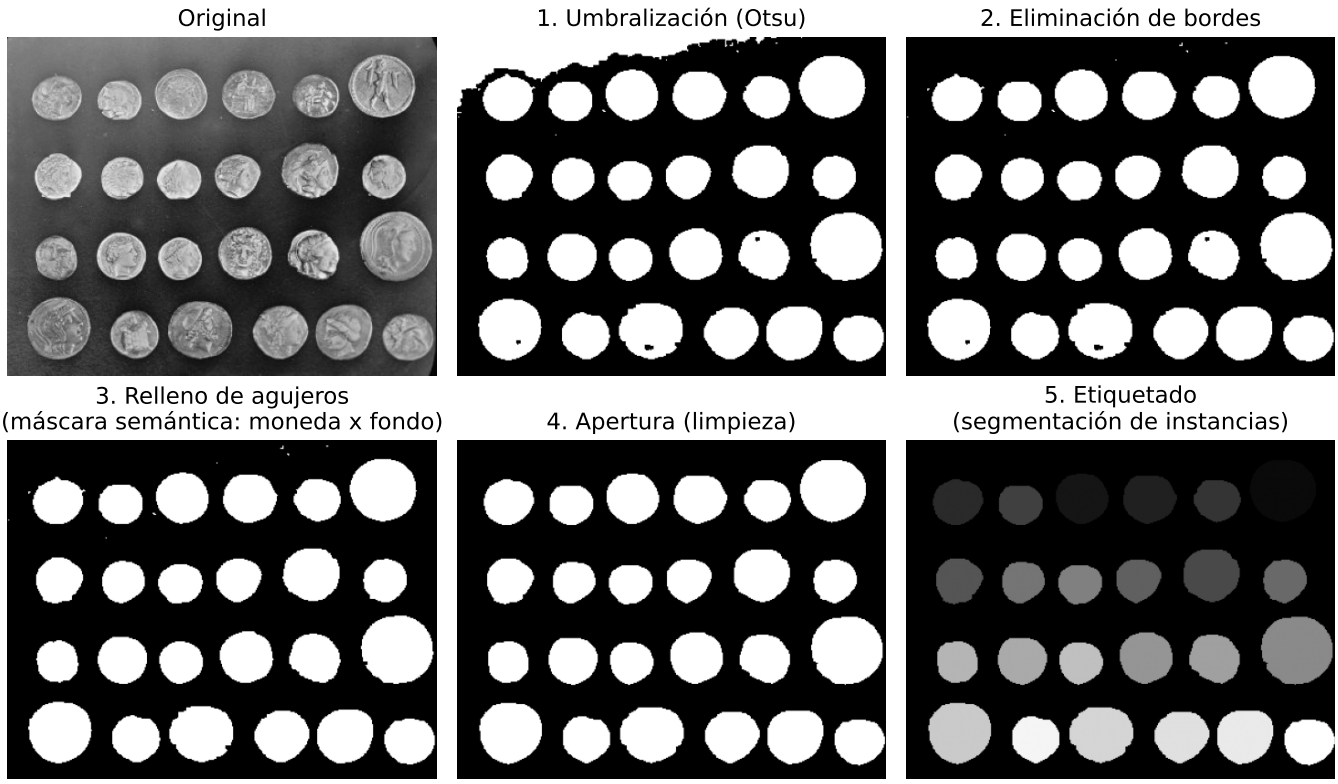


Figura 8.12: Segmentación clásica (no basada en aprendizaje profundo): máscara semántica (moneda x fondo) mediante umbralización de Otsu, y segmentación de instancias mediante etiquetado de componentes conectados.

> ℹ 🧠 ¿Por qué funciona? — Y dónde el enfoque clásico encuentra limitaciones
>
> En este ejemplo, la segmentación se facilita por el contraste entre las monedas y el fondo, así como por la relativa homogeneidad de intensidad en el interior de cada moneda. La umbralización de Otsu separa eficientemente estas dos regiones, mientras que las operaciones morfológicas eliminan pequeñas imperfecciones de la máscara binaria. Finalmente, el etiquetado de componentes conectados asigna un identificador distinto a cada región conexa, produciendo una segmentación de instancias.
> Esta estrategia, sin embargo, depende directamente de la calidad de la máscara binaria. Si dos objetos están unidos, superpuestos o presentan un contraste insuficiente respecto al fondo, pueden ser representados por una única región o dejar de ser segmentados correctamente. Además, los métodos basados predominantemente en la intensidad de los píxeles tienen una capacidad limitada para distinguir objetos de diferentes clases con apariencia similar.
> Los métodos modernos de segmentación basados en redes neuronales profundas aprenden representaciones visuales directamente a partir de los datos de entrenamiento, lo que generalmente les confiere una mayor capacidad para manejar variaciones de iluminación, textura, forma y oclusión. La segmentación panóptica amplía este enfoque al combinar, en una única representación, la clasificación semántica de todos los píxeles y la identificación individual de los objetos presentes en la escena.

8.8.3 Alternativa: Transformada de Distancia + *Watershed*

La apertura morfológica puede separar monedas que se tocan mediante la erosión de la máscara binaria. Sin embargo, esta operación modifica el contorno de todos los objetos, incluidos aquellos que ya estaban aislados. Una alternativa, presentada en el **Capítulo 4**, consiste en combinar la **transformada de distancia** con el algoritmo ***watershed***, utilizando marcadores obtenidos a partir de la propia máscara binaria.

El procedimiento se realiza en tres etapas:

1. se calcula la **transformada de distancia** de la máscara binaria, asignando a cada píxel del objeto su distancia hasta el fondo más cercano;
2. se aplica un umbral a la transformada de distancia para obtener **marcadores** situados en las regiones centrales de las monedas;
3. se utiliza el algoritmo ***watershed*** para expandir estos marcadores hasta las fronteras entre los objetos, separando monedas que se encuentran en contacto.

La Figura 8.13 presenta estas etapas. Los marcadores se obtienen en las regiones centrales de la transformada de distancia y se utilizan como semillas para el *watershed*, que propaga cada etiqueta hasta encontrar las fronteras entre objetos vecinos.

Comparando con la apertura morfológica, ambos enfoques logran separar monedas en contacto. Sin embargo, como el *watershed* utiliza marcadores para dividir regiones conectadas, sin aplicar erosión directamente a la máscara, los contornos originales tienden a preservarse mejor, favoreciendo la obtención de medidas geométricas, como área, perímetro y circularidad.

```
# Reutiliza la máscara semántica ANTES de la apertura agresiva (sin erosión del contorno)
mascara_base = mascara_semantica

# 1. Transformada de distancia: cada píxel de la máscara recibe la distancia hasta el fondo
cercano
distancia = mm.dist(mascara_base)

# Marcadores obtenidos por umbralización de la transformada de distancia.
# Las regiones centrales de las monedas permanecen conectadas y se utilizan
# como semillas para el algoritmo watershed.
marcadores = mm.label(
    np.uint8(distancia > 0.5 * distancia.max())
)

# 3. Watershed: propaga cada marcador dentro de la máscara hasta los puntos de contacto entre
monedas
rotulos_watershed = mm.watershed(marcadores, mascara_base)

print(f"Apertura morfológica (mascara_limpa): {np.max(rotulos_instancias)} instancias")
print(f"Distancia + watershed: {np.max(rotulos_watershed)} instancias")

mm.show(
    [mascara_base, distancia, mm.dil(marcadores,mm.sedisk(5)), rotulos_watershed],
    titles=[
        "Máscara semántica\n(sin apertura, sin erosión)",
        "Transformada de distancia",
        "Marcadores\n(máximos regionales)",
        "Watershed\n(instancias separadas)"
    ],
    cols=4, figsize=(11, 3.2)
)
```

```
Apertura morfológica (mascara_limpa): 24 instancias
Distancia + watershed: 24 instancias
```

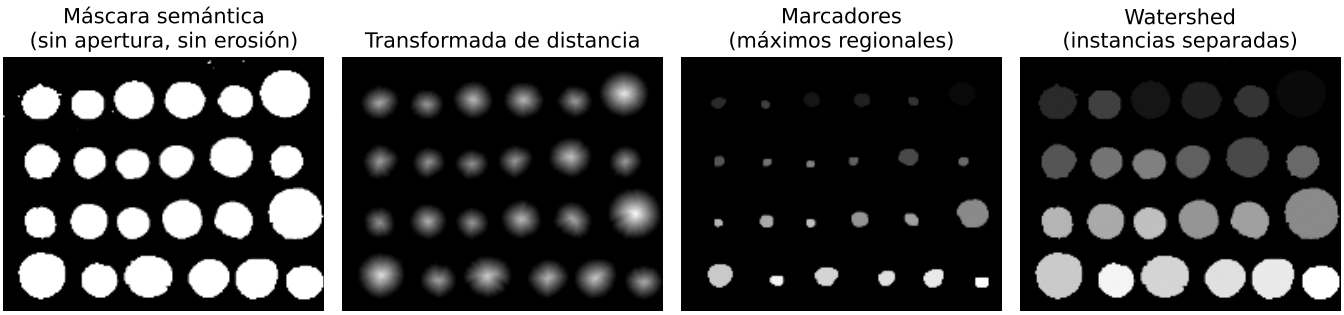


Figura 8.13: Separación de monedas que se tocan mediante transformada de distancia + *watershed*, sin erosión del contorno de las monedas.

> ¿Por qué funciona? — Marcadores y *Watershed*
>
> La transformada de distancia asigna valores mayores a los píxeles más alejados del fondo, que normalmente se encuentran en las regiones más centrales de los objetos. Al aplicar un umbral a esta transformada, se obtienen marcadores localizados en el interior de cada moneda, favoreciendo la obtención de un marcador para cada objeto.
> El algoritmo ***watershed*** utiliza estos marcadores como semillas y propaga sus etiquetas hasta que dos regiones en crecimiento se encuentran. Los puntos de encuentro entre estas regiones definen las fronteras entre objetos adyacentes.
> El rendimiento de este método depende de la calidad de los marcadores. Objetos muy alargados, con formas irregulares o que contienen múltiples máximos en la transformada de distancia pueden generar marcadores adicionales, resultando en una segmentación excesiva (*oversegmentation*). En aplicaciones más complejas, los métodos modernos de segmentación de instancias basados en redes neuronales profundas aprenden directamente, a partir de los datos de entrenamiento, representaciones adecuadas para separar los objetos, prescindiendo de la construcción explícita de marcadores y otras heurísticas geométricas.

8.8.4 Medición y Extracción de Atributos con `mm.measure`

Tras la segmentación y el etiquetado de los objetos, el siguiente paso consiste en medir sus propiedades geométricas. Para ello, la biblioteca `morph` ofrece la función `mm.measure`, que recibe una imagen **binaria** y devuelve, para cada objeto, un conjunto de descriptores geométricos organizado en una lista de diccionarios.

Entre los descriptores calculados destacan el **área**, el **perímetro**, el **centroide**, la **caja delimitadora** (*bounding box*), la **circularidad**, la **solidez** y el número de **vértices** del contorno aproximado. Estos vértices se obtienen aproximando el contorno mediante un polígono simplificado, calculado por la función `approxPolyDP`, que preserva la forma general del objeto utilizando un número reducido de segmentos.

La circularidad se define como

$$\text{Circularidad} = \frac{4\pi \cdot \text{Área}}{\text{Perímetro}^2},$$

asumiendo un valor igual a 1 para un círculo perfecto y valores menores para formas progresivamente menos circulares.

La solidez viene dada por

$$\text{Solidez} = \frac{\text{Área}}{\text{Área del Cierre Convexo}},$$

indicando cuánto llena el objeto su cierre convexo. Valores cercanos a 1 caracterizan objetos convexos, mientras que valores menores indican la presencia de concavidades.

El número de vértices proporciona una indicación de la complejidad de la forma del objeto. Por ejemplo, un triángulo tiende a producir tres vértices y un rectángulo cuatro, mientras que objetos de contorno curvo, como monedas, suelen resultar en polígonos con un mayor número de vértices, dependiendo de la precisión adoptada en la aproximación.

Como las monedas de este ejemplo presentan contornos aproximadamente circulares y convexos, se espera que la circularidad asuma valores elevados (típicamente entre 0,8 y 0,9) y que la solidez permanezca muy cercana a 1. El número de vértices depende del parámetro utilizado en la aproximación poligonal (`precision`), aumentando a medida que la aproximación preserva más detalles del contorno. En conjunto, estos descriptores pueden emplearse en la clasificación de objetos, en la identificación de componentes espurios remanentes de la segmentación y en la realización de mediciones cuantitativas sobre la imagen.

```
# mm.measure recibe una imagen binaria; cada componente conexo corresponde
# a una moneda individual.
medidas = mm.measure(mascara_limpa, precision=0.01)

if medidas:
    print(f"Total de objetos catalogados: {len(medidas)}")
    print(f"{'ID':<4}{'Área':<10}{'Perímetro':<12}{'Circularidade':<14}",end="")
    print(f" {'Solidez':<10}{'Vértices':<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 se devolvió ninguna medida.")
```

```
Total de objetos catalogados: 24
ID  Área      Perímetro   Circularidade  Solidez   Vértices
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
```

Se observa que se identificaron **24 monedas**. Las medidas de solidez permanecen cercanas a 1, indicando contornos esencialmente convexos, mientras que la circularidad varía entre aproximadamente 0,84 y 0,90 debido a las irregularidades del contorno discretizado. El número de vértices varía entre 9 y 14, reflejando la aproximación poligonal utilizada para representar cada moneda.

8.8.5 Exportación de Anotaciones: de `mm.measure` al Formato YOLO

Además de los descriptores geométricos, `mm.measure` calcula, para cada objeto, su **caja delimitadora** (*bounding box*). Esta información puede exportarse automáticamente mediante la función `mm.saveMeasures`, lo que permite generar archivos de anotación para diferentes aplicaciones sin necesidad de etiquetado manual.

La función admite tres formatos de salida:

- `fmt="csv"`: exporta todos los descriptores geométricos, siendo útil para análisis exploratorio, medición y clasificación basada en atributos;
- `fmt="txt"`: guarda los descriptores en formato tabular simple;
- `fmt="yolo"`: genera anotaciones compatibles con el formato utilizado por los detectores de la familia **YOLO** (*You Only Look Once*).

En el formato YOLO, cada objeto se representa mediante una línea que contiene

$$\text{clase} \;\; x_c \;\; y_c \;\; w_n \;\; h_n,$$

donde (x_c, y_c) representa el centro de la caja delimitadora y (w_n, h_n) sus dimensiones, todos normalizados por las dimensiones de la imagen:

$$x_c = \frac{x + w/2}{W}, \qquad y_c = \frac{y + h/2}{H}, \qquad w_n = \frac{w}{W}, \qquad h_n = \frac{h}{H},$$

donde (x, y) corresponden a las coordenadas de la esquina superior izquierda de la caja delimitadora, (w, h) a sus dimensiones en píxeles y (W, H) al ancho y alto de la imagen. La normalización hace que las anotaciones sean independientes de la resolución de la imagen, permitiendo utilizar el mismo formato en imágenes de diferentes tamaños.

El siguiente código exporta las medidas extraídas de las monedas a los formatos **CSV** y **YOLO**. La salida en CSV preserva todos los descriptores geométricos calculados por `mm.measure`, mientras que el archivo en formato YOLO contiene únicamente la clase y la caja delimitadora normalizada de cada objeto, según lo requerido por los detectores de esta familia.

Esta representación se utilizará nuevamente en el **Capítulo 9**, dedicado a los métodos de detección de objetos basados en *Deep Learning*.

```python
# Exporta las medidas extraídas de las monedas en dos formatos de anotación
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("--- Fragmento de moedas.csv ---")
with open("moedas.csv") as f:
    for linha in f.readlines()[:4]:
        print(linha.rstrip())

print("\n--- Fragmento de moedas.txt (formato YOLO) ---")
with open("moedas.txt") as f:
    for linha in f.readlines()[:4]:
        print(linha.rstrip())
```

```
--- Fragmento de moedas.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
```

```
--- Fragmento de moedas.txt (formato YOLO) ---
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 Validação de Anotaciones por IoU: `mm.verifyBoundBox`

Antes de utilizar anotaciones en el entrenamiento o en la evaluación de modelos, es importante verificar si estas concuerdan con un conjunto de referencia (*ground truth*).

La función `mm.verifyBoundBox` realiza esta comparación utilizando la métrica **IoU** (*Intersection over Union*), presentada anteriormente en este capítulo. Para cada caja delimitadora candidata, la función calcula su superposición con las cajas del patrón de referencia y contabiliza aquellas cuyo IoU es superior a un umbral especificado.

En el siguiente ejemplo, se construye un **patrón de referencia sintético** a partir de las cinco primeras cajas delimitadoras obtenidas por `mm.measure`. Como las anotaciones comparadas son las mismas utilizadas para construir el patrón de referencia, se espera que cada objeto encuentre exactamente una correspondencia con IoU $\geq 0{,}5$. El objetivo es únicamente ilustrar el funcionamiento de la función `mm.verifyBoundBox`; en aplicaciones reales, el patrón de referencia debe obtenerse de manera independiente, por ejemplo, mediante anotación manual.

```python
# Matriz de plantillas sintética en el formato (clase, x1, y1, x2, y2), normalizado
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"Objeto {m['id']}: {correspondencias} plantilla(s) coinciden con IoU >= 0.5")

print(f"\nTotal de objetos validados: {acertos}/{len(medidas[:5])}")
```

```
Objeto 1: 1 plantilla(s) coinciden con IoU >= 0.5
Objeto 2: 1 plantilla(s) coinciden con IoU >= 0.5
Objeto 3: 1 plantilla(s) coinciden con IoU >= 0.5
Objeto 4: 1 plantilla(s) coinciden con IoU >= 0.5
Objeto 5: 1 plantilla(s) coinciden con IoU >= 0.5

Total de objetos validados: 5/5
```

Se observa que los cinco objetos fueron correctamente asociados a sus respectivas plantillas, resultando en cinco correspondencias válidas. En aplicaciones reales, esta misma estrategia puede emplearse para comparar automáticamente las cajas delimitadoras producidas por un detector con anotaciones de referencia, permitiendo evaluar cuantitativamente su desempeño mediante la métrica IoU.

8.8.7 Visualización de Anotaciones Guardadas: `mm.showBoundBox`

Después de exportar las anotaciones, es conveniente verificar si las cajas delimitadoras fueron guardadas correctamente. Para ello, la biblioteca `morph` pone a disposición la función `mm.showBoundBox`, ilustrada en la Figura 8.14.

La función lee un archivo de anotaciones en los formatos `yolo`, `csv` o `txt`, reconstruye las **cajas delimitadoras** (*bounding boxes*) y las superpone a la imagen original, facilitando la inspección visual del resultado. De esta manera, es posible confirmar rápidamente si las anotaciones están alineadas con los objetos, sin necesidad de examinarlas manualmente.

Esta función complementa a `mm.verifyBoundBox`. Mientras que `mm.verifyBoundBox` realiza una **validación cuantitativa**, comparando las anotaciones con un conjunto de referencia mediante la métrica IoU, `mm.showBoundBox` proporciona una **validación cualitativa**, permitiendo inspeccionar visualmente las cajas delimitadoras reconstruidas.

```
caminho_yolo = "moedas.txt"
img_anotada = mm.showBoundBox(img_moedas, caminho_yolo, fmt="yolo")
```

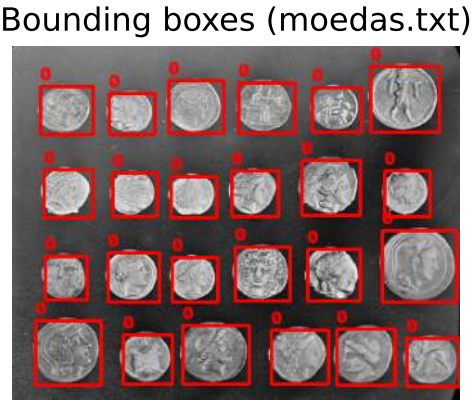


Figura 8.14: *Bounding boxes* reconstruidas a partir del archivo de anotaciones YOLO exportado por *mm.saveMeasures*, superpuestas a la imagen original de las monedas.

> ¿Por qué funciona? — De la Segmentación a las Anotaciones
>
> Tras segmentar una imagen, es posible medir automáticamente cada objeto (`mm.measure`) y convertir esas medidas en anotaciones (`mm.saveMeasures`) para diferentes formatos, incluido el utilizado por los detectores de la familia YOLO. Este procedimiento permite generar automáticamente conjuntos iniciales de anotaciones en escenarios controlados, reduciendo significativamente el trabajo de etiquetado manual.
> Antes de utilizar estas anotaciones en el entrenamiento o la evaluación de modelos, se recomienda compararlas con un conjunto de referencia (*ground truth*). La función `mm.verifyBoundBox` automatiza esta etapa mediante la métrica **IoU**, permitiendo cuantificar la concordancia entre las cajas delimitadoras generadas y las anotaciones de referencia. Complementariamente, `mm.showBoundBox` posibilita una inspección visual de las anotaciones reconstruidas sobre la imagen original, facilitando la identificación de errores de normalización, posicionamiento u orden de las coordenadas que pueden no ser evidentes en una validación únicamente numérica.
> En conjunto, `mm.measure`, `mm.saveMeasures`, `mm.verifyBoundBox` y `mm.showBoundBox` implementan un flujo completo para medir objetos segmentados, generar anotaciones, evaluarlas e inspeccionarlas visualmente. Cuando la segmentación produce resultados confiables, este flujo puede reducir significativamente la necesidad de etiquetado manual en la construcción de conjuntos de datos.

8.9 Visión General de los Modelos Modernos

Las técnicas presentadas en este capítulo — como detectores en cascada, segmentación basada en umbralización y posprocesamiento mediante IoU y NMS — siguen siendo importantes para comprender los fundamentos de la Visión por Computadora. Actualmente, sin embargo, muchas aplicaciones utilizan modelos de **Aprendizaje Profundo**, capaces de aprender automáticamente representaciones discriminativas a partir de grandes conjuntos de datos, prescindiendo de la definición manual de características.

La Tabla 8.2 presenta algunas arquitecturas representativas para la detección y segmentación de imágenes. El objetivo es solo situarlas en el contexto de las tareas estudiadas en este capítulo. Sus principios de funcionamiento, entrenamiento y aplicación se discutirán en detalle en el **Capítulo 9**.

Tabla 8.2: Arquitecturas representativas de Aprendizaje Profundo para la detección y segmentación de imágenes.

Modelo	Tarea	Idea central
YOLO (*You Only Look Once*)	Detección de objetos	Detecta objetos en una sola pasada por la red, estimando clases y cajas delimitadoras. Versiones recientes, como el **YOLO26** (2026), eliminan la etapa de *Non-Maximum Suppression* (NMS), haciendo que la inferencia sea totalmente *end-to-end*.
Faster R-CNN	Detección de objetos	Genera regiones candidatas y las refina antes de la clasificación, priorizando la precisión.
SSD (*Single Shot Detector*)	Detección de objetos	Detecta objetos en diferentes escalas en una sola pasada, buscando un equilibrio entre velocidad y precisión.
U-Net	Segmentación semántica	Produce una máscara que clasifica cada píxel de la imagen según su clase.
Mask R-CNN	Segmentación de instancias	Extiende el Faster R-CNN añadiendo una máscara individual para cada objeto detectado.
***Segment Anything* (SAM)**	Segmentación orientada por *prompts*	Segmenta objetos a partir de indicaciones como puntos, cajas delimitadoras o máscaras. Evolucionó hacia el **SAM 2**, con soporte para video y seguimiento temporal de objetos, y el **SAM 3**, que permite una segmentación guiada directamente por descripciones en texto.

Se observa una evolución de las tareas abordadas en este capítulo. Modelos como **YOLO**, **Faster R-CNN** y **SSD** localizan objetos mediante cajas delimitadoras. El **Mask R-CNN** amplía esta capacidad al producir también una máscara para cada instancia detectada. Por su parte, modelos más recientes, como el ***Segment Anything* (SAM)**, permiten segmentar objetos a partir de diferentes tipos de *prompts*, haciendo que el proceso sea más flexible.

Esta evolución acompaña la secuencia de conceptos desarrollada a lo largo del capítulo: desde la correspondencia de características y la localización aproximada mediante cajas delimitadoras hasta la segmentación precisa de los píxeles pertenecientes a cada objeto. En el **Capítulo 9**, estas tareas se revisitarán desde la perspectiva del Aprendizaje Profundo, explorando cómo las redes neuronales modernas aprenden automáticamente representaciones capaces de superar muchas de las limitaciones de los métodos clásicos presentados aquí.

8.10 Limitaciones de los Enfoques Clásicos y Motivación para el *Deep Learning*

Los ejemplos desarrollados en este capítulo ilustran tres problemas fundamentales de la Visión Computacional: **correspondencia de características**, **detección de objetos** y **segmentación de imágenes**. Aunque las técnicas clásicas presentadas son eficientes en diversos escenarios, todas comparten una limitación importante: se basan en características definidas manualmente para describir o identificar los objetos de interés.

- El **ORB** utiliza descriptores binarios capaces de establecer correspondencias entre puntos bajo rotaciones y cambios moderados de escala, pero no fue concebido para reconocer categorías de objetos.
- El **Haar Cascade** emplea un conjunto fijo de filtros rectangulares, funcionando bien para objetos con apariencia relativamente estandarizada, como rostros frontales, pero volviéndose más susceptible a falsos positivos en escenas complejas.
- La **segmentación por umbralización** explora diferencias de intensidad entre objeto y fondo, siendo adecuada para imágenes con buen contraste, aunque limitada ante variaciones de iluminación, texturas o escenas que contienen múltiples clases de objetos.

En todos estos casos, el rendimiento depende de la capacidad de las características elegidas para representar adecuadamente la variabilidad presente en las imágenes. En aplicaciones reales, factores como cambios de pose, iluminación, escala, oclusiones y diversidad de categorías hacen que esta tarea sea cada vez más difícil, reduciendo la generalización de los métodos clásicos.

El **Capítulo 9** presenta un enfoque diferente para estos problemas mediante las **Redes Neuronales Convolucionales (CNNs)**. En lugar de utilizar características definidas manualmente, estos modelos aprenden automáticamente, a partir de grandes conjuntos de datos, representaciones adecuadas para cada tarea. Las arquitecturas modernas presentadas en la sección anterior —como **YOLO**, **Faster R-CNN**, **U-Net**, **Mask R-CNN** y ***Segment Anything*** **(SAM)**— siguen este principio y representan una evolución de las técnicas clásicas estudiadas en este capítulo, ampliando su capacidad para manejar la diversidad y la complejidad de las imágenes del mundo real.

8.11 Resumen

En este capítulo se estudiaron métodos clásicos de Visión por Computador para **correspondencia de características**, **detección de objetos** y **segmentación de imágenes**. Al final, se presentaron algunas arquitecturas modernas basadas en Aprendizaje Profundo, preparando la transición hacia el **Capítulo 9**. Los principales conceptos abordados fueron:

- **ORB:** detección de puntos clave, construcción de descriptores binarios y correspondencia de características mediante la distancia de Hamming.
- **Homografía y RANSAC:** estimación robusta de transformaciones proyectivas para el registro de imágenes y eliminación de correspondencias inconsistentes.
- **Haar Cascade:** detección de objetos utilizando características Haar, imagen integral y clasificadores en cascada.
- **Segmentación clásica:** umbralización, operaciones morfológicas, etiquetado de componentes conectados, transformada de distancia y algoritmo *watershed* para la segmentación de instancias.
- **Medición y anotación de objetos:** extracción de atributos geométricos con `mm.measure`, exportación automática de anotaciones (`mm.saveMeasures`) y validación mediante IoU (`mm.verifyBoundBox`) e inspección visual (`mm.showBoundBox`).

- **Evaluación de detecciones:** uso de la métrica **Intersection over Union (IoU)** y de la **Supresión de No Máximos (NMS)**.
- **Modelos modernos:** visión general de las arquitecturas **YOLO**, **Faster R-CNN**, **SSD**, **U-Net**, **Mask R-CNN** y **Segment Anything (SAM)**, que servirán de base para el estudio de los métodos de Aprendizaje Profundo en el **Capítulo 9**.

Próximos Pasos

En este capítulo se presentaron métodos clásicos para resolver problemas de **correspondencia de características**, **detección de objetos** y **segmentación de imágenes**, basados en descriptores, filtros y modelos geométricos definidos manualmente.

En el **Capítulo 9**, estos mismos problemas serán revisitados desde la perspectiva del **Aprendizaje Profundo**, con énfasis en las **Redes Neuronales Convolucionales (CNNs)**. En lugar de utilizar características diseñadas manualmente, estos modelos aprenden automáticamente representaciones a partir de grandes conjuntos de datos, constituyendo la base de los principales sistemas modernos de Visión Computacional para detección, segmentación y reconocimiento de objetos.

8.12 🤖 Uso de Gemini Notebook como Apoyo al Estudio

Gemini Notebook puede utilizarse como herramienta complementaria para revisar los conceptos presentados en este capítulo. A partir del contenido disponible como referencia, permite responder preguntas, elaborar resúmenes, aclarar dudas y explorar los temas de forma interactiva.

Acceso al Notebook del Capítulo

🚀 ACCEDER A GEMINI NOTEBOOK: CAPÍTULO 08

🌐 **Idioma y Lenguaje de Programación**

El proyecto de este capítulo en Gemini Notebook fue construido únicamente con el texto en **portugués** y los ejemplos de código en **Python**. Si estás estudiando la edición en inglés o francés, o siguiendo la ruta en C++, las respuestas del tutor pueden no corresponder exactamente con la versión que estás leyendo.
Las respuestas generadas por Gemini Notebook se producen automáticamente y pueden contener imprecisiones. Úsalas como material complementario de estudio, contrastando la información con el contenido de este libro y, cuando sea necesario, con otras fuentes académicas.

8.13 Lista de Ejercicios

Los ejercicios siguientes consolidan los conceptos presentados en este capítulo mediante adaptaciones, experimentos y extensiones de los algoritmos desarrollados a lo largo del texto, utilizando la biblioteca didáctica `morph`.

1. **(10%)** Investigue la influencia de las rotaciones en la estabilidad del detector y descriptor **ORB**. Utilizando la imagen `skimage.data.astronaut()`, genere versiones rotadas en ángulos de $0°, 45°, 90°, 135°, 180°$. Para cada caso, determine el número de correspondencias obtenidas por el `BFMatcher` y la fracción de *inliers* identificados por el **RANSAC**. Presente los resultados en una tabla y discuta la robustez del método frente a las rotaciones analizadas.

2. **(15%)** Compare el detector de esquinas **FAST** con el detector de **Harris** (Capítulo 6). Mida el tiempo medio de ejecución y el número de puntos detectados en las imágenes `skimage.data.camera()`, `skimage.data.gravel()` y `skimage.data.brick()`. Discuta las ventajas y limitaciones de cada detector en términos de velocidad y repetibilidad.

3. **(15%)** Investigue la sensibilidad del **Haar Cascade** a la presencia de ruido y desenfoque. Añada ruido gaussiano ($\sigma \in 5, 10, 20$) y desenfoque de movimiento a la imagen `skimage.data.astronaut()`. Varíe los parámetros `scaleFactor` y `minNeighbors` de la función `detectMultiScale()` y discuta los efectos sobre falsos positivos, falsos negativos y el número de caras detectadas.

4. **(15%)** Implemente manualmente la métrica **Intersection over Union (IoU)** para cajas delimitadoras y compare sus resultados con la función `mm.IoU`. A continuación, cree una caja de referencia y diez cajas obtenidas mediante desplazamientos y variaciones de escala, construyendo un gráfico que relacione el desplazamiento aplicado con el valor de IoU obtenido.

5. **(15%)** Evalúe el algoritmo de **Supresión de No Máximos (NMS)**. Cree una escena sintética que contenga tres objetos, cada uno representado por al menos cinco cajas superpuestas con diferentes puntuaciones de confianza. Ejecute el algoritmo para umbrales de IoU iguales a $0.2, 0.4, 0.6, 0.8$, visualice los resultados y discuta la influencia de este parámetro en la cantidad de cajas mantenidas.

6. **(15%)** Aplique el pipeline de segmentación presentado en este capítulo (umbralización de Otsu, operaciones morfológicas y `mm.measure`) a la imagen `skimage.data.coins()`. Exporte las medidas utilizando `mm.saveMeasures(fmt="csv")` y produzca histogramas de las distribuciones de área, circularidad y solidez. Discuta cuáles de estos atributos son más adecuados para caracterizar las monedas.

7. **(15%)** Construya una imagen sintética que contenga círculos, rectángulos y triángulos, algunos parcialmente superpuestos. Realice la segmentación, extraiga las medidas con `mm.measure` y genere automáticamente las anotaciones en formato YOLO utilizando `mm.saveMeasures(fmt="yolo")`. A continuación, modifique deliberadamente algunas *bounding boxes* y utilice `mm.verifyBoundBox` para evaluar, en diferentes umbrales de IoU, cuántas anotaciones permanecen válidas. Discuta la influencia de la superposición entre objetos en la calidad de las anotaciones obtenidas.

8. **(Bono – 10%)** Desarrolle un clasificador simple, como **k-NN** (Capítulo 7), utilizando únicamente los atributos geométricos producidos por `mm.measure` (área, circularidad, solidez y número de vértices). Utilice un conjunto de formas sintéticas (círculos, rectángulos y triángulos) con diferentes escalas y rotaciones, evalúe la precisión obtenida y discuta la capacidad de estos descriptores para distinguir diferentes clases de objetos.

Referencias del Capítulo

Los conceptos y algoritmos presentados en este capítulo se fundamentaron en referencias clásicas y contemporáneas de la literatura de PDI-VC:

- Gonzalez (2018), para los fundamentos de segmentación de imágenes, operaciones morfológicas, etiquetado de componentes conexos y extracción de atributos geométricos.
- Szeliski (2022), para los conceptos de correspondencia de características, transformaciones geométricas, homografías, detección de objetos y segmentación en Visión por Computador.
- Rublee (2011), para la descripción del algoritmo **ORB** (*Oriented FAST and Rotated BRIEF*), utilizado en la detección, descripción y correspondencia de características locales.
- Fischler (1981), para la formulación original del algoritmo **RANSAC** (*Random Sample Consensus*), empleado en la estimación robusta de modelos geométricos en presencia de datos inconsistentes.
- Viola (2001), Viola (2004) y Lienhart (2002), para los fundamentos del detector **Haar Cascade**, incluyendo características de Haar, imagen integral, clasificadores en cascada y extensiones utilizadas en implementaciones modernas.
- Kirillov (2019), para la definición de la **segmentación panóptica**, que integra segmentación semántica y segmentación de instancias en una única representación.
- Redmon (2016), Ren (2015), Ronneberger (2015), He (2016) y Kirillov (2019), para modelos modernos basados en Aprendizaje Profundo aplicados a la detección y segmentación de imágenes, incluyendo **YOLO**, **Faster R-CNN**, **U-Net**, **Mask R-CNN** y **Segment Anything (SAM)**.

Executar Colab | Abrir GitHub

8.14 💻 Parte Práctica con Ejercicios de Programación

La presente lista de ejercicios de programación (EP) consolida las formulaciones teóricas presentadas a lo largo del Capítulo 8 — Correspondencia de Características, Detección de Objetos y Segmentación Clásica — mediante una ruta práctica aplicada. Así como en el capítulo anterior, los EPs aíslan las **magnitudes intermedias** de cada técnica — la distancia entre descriptores binarios, los términos de una imagen integral, el conteo de *inliers* de un modelo candidato, la superposición entre cajas delimitadoras y la etiqueta de cada componente conectado — permitiendo validar manualmente cada etapa del razonamiento sin depender de OpenCV ni de imágenes externas.

El encadenamiento de los ejercicios reproduce el flujo conceptual del capítulo: se inicia con la **distancia de Hamming**, corazón de la correspondencia de descriptores binarios como el ORB; se avanza hacia el conteo de ***inliers*** que sustenta el **RANSAC** en la estimación robusta de una homografía; se continúa con la **imagen integral**, el truco computacional que hace viable al ***Haar Cascade*** en tiempo real; se profundiza en **IoU y Supresión de No-Máximos**, el post-procesamiento común a prácticamente todo detector de objetos; y se concluye con el **etiquetado de componentes conectados**, el enfoque clásico — y sus limitaciones — para segmentar instancias individuales en una máscara binaria.

🎯 Objetivo de este Cuaderno

El cuaderno permite desarrollar, validar, organizar y probar soluciones de **Ejercicios de Programación (EPs)** en entornos interactivos, como Colab, con los mismos casos de prueba de Moodle, copiándolos allí solo al momento de registrar la nota oficial.

Download

Descargue `morph.py` y `testsuite.py` ejecutando la celda siguiente:

```
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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Ejecutando las pruebas

Para evaluar las pruebas, ejecute `TestSuite("EP08_01.extensión").run()` en una nueva celda, reemplazando la extensión por la del lenguaje utilizado (`.py`, `.java`, `.c`, `.cpp`, `.js` o `.r`). El sistema descarga los casos de prueba de GitHub, ejecuta el programa y calcula la nota automáticamente.

Para probar código Python directamente, sin guardar un archivo, use `run_code(codigo)` pasando el código como *string* en una variable `codigo`:

```
codigo = """
# ... su código aquí ...
"""
TestSuite("EP08_01").run_code(codigo)
```

Resumen de los Métodos de `morph.py` (Cap. 8)

La biblioteca `morph.py` proporciona funciones para el análisis de componentes conexos, extracción de contornos, métricas geométricas y anotaciones:

1. **Componentes y Contornos (`connectedComponents`, `findContours`)** Etiquetan regiones conexas y extraen los contornos de imágenes binarias.
2. **Propiedades del Contorno (`contourArea`, `arcLength`, `convexHull`, `approxPolyDP`, `fitLine`)** Calculan área, perímetro, envolvente convexa, aproximación poligonal y ajuste de recta para un contorno.
3. **Geometría Envolvente (`boundingRect`, `minAreaRect`, `boxPoints`, `minEnclosingCircle`, `fitEllipse`)** Determinan rectángulos envolventes (alineados u orientados), elipses y el círculo delimitador más pequeño.
4. **Extracción y Persistencia de Medidas (`measure`, `saveMeasures`)** Extraen descriptores geométricos de los objetos (área, circularidad, solidez, centroide) y exportan los datos a CSV, texto o formato YOLO.
5. **Evaluación y Visualización (`IoU`, `verifyBoundBox`, `showBoundBox`)** Calculan la intersección sobre unión (*Intersection over Union*), validan cajas delimitadoras con patrones de referencia y dibujan cajas delimitadoras anotadas sobre la imagen.

8.14.1 EP08_01 🟢 Distancia de Hamming y Correspondencia de Descriptores Binarios

ORB, utilizado en el Proyecto Práctico 1 de este capítulo, describe la vecindad de cada punto de interés como una secuencia de bits — y, por ello, la comparación entre dos descriptores no utiliza la distancia euclidiana del k-NN del Capítulo 7, sino la **distancia de Hamming**: el número de posiciones en que los bits difieren. Antes de llamar a `cv2.BFMatcher(cv2.NORM_HAMMING)`, se te encargó implementar manualmente esta correspondencia (*matching*) por fuerza bruta — la misma etapa que, ejecutada internamente por OpenCV, precede a la estimación robusta de la homografía mediante RANSAC.

8.14.1.1 📋 Directrices de Implementación

1. **Cantidades:** Leer los enteros N y M — número de descriptores extraídos de la imagen A y de la imagen B, respectivamente.
2. **Descriptores de A:** Leer N líneas, cada una conteniendo un descriptor binario (una *cadena* de caracteres `0` y `1`, todos de la misma longitud).
3. **Descriptores de B:** Leer M líneas, en el mismo formato.
4. **Umbral:** Leer el entero τ — distancia de Hamming máxima aceptable para considerar una correspondencia válida.
5. **Distancia de Hamming:** Para dos descriptores binarios a y b de igual longitud,

$$d_H(a, b) = \sum_k \mathbb{1}[a_k \neq b_k],$$

   es decir, el conteo de posiciones en que los bits difieren.
6. **Correspondencia por vecino más cercano:** Para cada descriptor a_i de A (i en el orden de lectura, comenzando en 0), calcula su distancia de Hamming a **todos** los descriptores de B y encuentra el de menor distancia. En caso de empate entre dos o más descriptores de B con la misma distancia mínima, elige el de **menor índice**.
7. **Filtrado por umbral:** Si la menor distancia encontrada es $\leq \tau$, la correspondencia es válida; de lo contrario, a_i no posee correspondencia.
8. **Salida:** Para cada i de 0 a $N-1$, en el orden de lectura, imprimir una línea: `i j d` si hay correspondencia válida (donde j es el índice del descriptor de B elegido y d su distancia), o `i -1` en caso contrario. Al final, imprimir `Total correspondencias válidas: X`.

8.14.1.2 📌 Restricciones Computacionales

- **Misma longitud:** todos los descriptores (de A y de B) tienen exactamente el mismo número de bits.

- **Fuerza bruta:** compara cada descriptor de A con **todos** los de B — no se necesita ningún tipo de indexación ni estructura de aceleración.
- **Desempate por menor índice en B**, y **nunca** por orden de lectura de A (que ya es natural, pues cada a_i se trata de forma independiente).

8.14.1.3 🧠 Fundamentación Teórica

Elemento	Papel en la correspondencia ORB
Descriptor binario (BRIEF)	Cada bit es el resultado de una comparación de intensidad entre dos píxeles de la vecindad
Distancia de Hamming	Métrica de disimilitud entre *cadenas* binarias; mucho más rápida de calcular que la distancia euclidiana (operación XOR + conteo de bits)
Vecino más cercano	Criterio de correspondencia: cada punto de A se empareja con el punto de B cuyo descriptor sea más similar
Umbral τ	Filtra correspondencias poco confiables incluso antes del RANSAC — pero, como se discutió en el capítulo, algunas correspondencias incorrectas aún pasan, exigiendo la robustez del RANSAC

8.14.1.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Enteros N y M.
- Siguientes N líneas: un descriptor binario por línea (*cadena* de `0`s y `1`s).
- Siguientes M líneas: un descriptor binario por línea, en el mismo formato.
- Última línea: Entero τ.

Salida:

- N líneas, una por descriptor de A, en el formato `i j d` o `i -1`.
- Última línea: `Total correspondencias válidas: X`.

8.14.1.5 📌 Ejemplos

Entrada	Salida	Observación
3 3 10101010 11110000 00001111 10101011 00001110 11111111 2	0 0 1 1 -1 2 1 1 Total correspondencias válidas: 2	El descriptor `11110000` no encuentra correspondencia: su vecino más cercano está a distancia 4, por encima del umbral $\tau = 2$.

```
%%writefile EP08_01.py
# Código Python
```

```
Overwriting EP08_01.py
```

Versión interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap08/sim-ep0801.html

Figura 8.15: Simulador EP08_01: Distancia de Hamming entre Dos Descriptores Binarios

```
TestSuite("EP08_01.py").run()
```

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

8.14.2 EP08_02 🟢 Homografía y RANSAC: La Votación por *Inliers*

El RANSAC, presentado en la sección "Modelado Matemático: Homografía y RANSAC", repite un ciclo de tres pasos — sortear una muestra mínima, estimar un modelo candidato y contar cuántas correspondencias son consistentes con él (los ***inliers***) — manteniendo al final el modelo más votado. La etapa de estimación del modelo a partir de 4 puntos (paso 2) involucra álgebra lineal que queda fuera del alcance de este EP; aquí, recibes directamente un conjunto de homografías **ya candidatas** — como si cada una hubiera sido estimada a partir de una muestra aleatoria diferente — y tienes la tarea de reproducir exactamente el paso decisivo del algoritmo: **aplicar cada modelo a todas las correspondencias y contar sus *inliers***, eligiendo al ganador.

8.14.2.1 📋 Directrices de Implementación

1. **Correspondencias:** Leer el entero N y, a continuación, N líneas con cuatro reales cada una, x y x' y' — un punto de la imagen A y su correspondiente (posiblemente incorrecto) en la imagen B, exactamente como lo produce la etapa de *matching* del EP08_01.
2. **Modelos candidatos:** Leer el entero K (número de homografías candidatas) y el real ε (umbral de error de reproyección). Luego, leer K líneas, cada una con nueve reales h_{11} h_{12} h_{13} h_{21} h_{22} h_{23} h_{31} h_{32} h_{33} — los elementos de la matriz H candidata, en orden de lectura por fila (*row-major*).
3. **Reproyección:** Para cada correspondencia (x, y, x', y') y cada modelo candidato H_k, calcular el punto proyectado
$$\begin{bmatrix}\hat{x}\\ \hat{y}\\ \hat{w}\end{bmatrix} = H_k \begin{bmatrix}x\\ y\\ 1\end{bmatrix}, \qquad (\hat{x}/\hat{w},\ \hat{y}/\hat{w}) \text{ es el punto proyectado.}$$
4. **Error de reproyección:** $e = \sqrt{(\hat{x}/\hat{w} - x')^2 + (\hat{y}/\hat{w} - y')^2}$.
5. **Conteo de *inliers*:** Una correspondencia es un *inlier* del modelo H_k si $e \leq \varepsilon$.
6. **Selección del mejor modelo:** El modelo ganador es el que tiene mayor número de *inliers*; en caso de empate, elige el de **menor índice** k (el primero encontrado durante el ciclo iterativo del RANSAC).
7. **Salida:** Para cada modelo k de 0 a $K-1$, en el orden de lectura, imprime `Modelo k: I inliers`. Al final, imprime `Mejor modelo: k_best con I_best inliers`.

8.14.2.2 📌 Restricciones Computacionales

- **Comparación inclusiva:** un error de reproyección **exactamente igual** a ε cuenta como *inlier* ($e \leq \varepsilon$).
- **Sin estimación de H:** las matrices ya se proporcionan listas — no es necesario (ni esperado) resolver ningún sistema lineal.

- **Empate resuelto por el menor índice**, reflejando el comportamiento natural de un algoritmo iterativo que recorre los modelos en orden y solo reemplaza al mejor encontrado hasta entonces cuando un nuevo modelo lo **supera estrictamente**.

8.14.2.3 🧠 Fundamentación Teórica

Elemento	Papel en el RANSAC
Muestra mínima (4 pares)	Suficiente para determinar los 8 grados de libertad de una homografía
Modelo candidato H_k	Estimado a partir de una muestra mínima; puede ser bueno o malo, dependiendo de si la muestra contenía *outliers*
Error de reproyección	Mide qué tan bien el modelo "predice" cada correspondencia observada
Inlier vs. *outlier*	Correspondencias consistentes con el modelo ganador (*inliers*) vs. las demás, típicamente correspondencias incorrectas del *matching*
Refinamiento final	En la práctica, tras elegir el mejor modelo, el RANSAC lo recalcula usando **solo** sus *inliers* — paso no exigido en este EP

8.14.2.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero N.
- Siguientes N líneas: cuatro reales x y x' y'.
- Siguiente línea: Entero K y real ε.
- Siguientes K líneas: nueve reales (elementos de H_k, *row-major*).

Salida:

- K líneas en el formato `Modelo k: I inliers`.
- Última línea: `Mejor modelo: k_best con I_best inliers`.

8.14.2.5 📌 Ejemplos

Entrada	Salida	Observación
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 Mejor modelo: 0 con 4 inliers	El Modelo 0 (escala $\times 2$) explica correctamente 4 de las 5 correspondencias; la 5ª, $(5,5) \to (1,1)$, es un *outlier* que ninguno de los dos modelos explica bien.

```
%%writefile EP08_02.py
# Código Python
```

```
Overwriting EP08_02.py
```

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap08/sim-ep0802.html

Figura 8.16: Simulador EP08_02: RANSAC — Votación por Inliers entre Modelos Candidatos

```
TestSuite("EP08_02.py").run()
```

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

8.14.3 EP08_03 🟢 Imagen Integral: Sumas Rectangulares en Tiempo Constante

Imagina una cámara de seguridad procesando 30 fotogramas por segundo, y para cada fotograma el sistema necesita recorrer la imagen en decenas de posiciones y escalas diferentes, probando en cada una un conjunto de características rectangulares para decidir "¿hay una cara aquí?". Si calcular la suma de intensidades de cada rectángulo requiriese sumar píxel a píxel, el sistema no tendría la menor oportunidad de evaluar en tiempo real — el cuello de botella estaría justamente en la parte más repetida del algoritmo. Es exactamente ese cuello de botella el que la imagen integral elimina.

El Haar Cascade evalúa miles de características rectangulares por ventana, en múltiples posiciones y escalas — algo inviable en tiempo real si cada rectángulo requiriese sumar sus píxeles uno a uno. La **imagen integral**, definida en la sección sobre Haar Cascade, resuelve este problema: una vez precomputada, la suma de intensidades de **cualquier** región rectangular se obtiene con solo cuatro consultas y tres operaciones aritméticas, independientemente del tamaño del rectángulo.

Se te ha encargado implementar esta estructura desde cero: primero, calcular la imagen integral a partir de la imagen original; luego, responder a consultas rectangulares arbitrarias.

8.14.3.1 📋 Directrices de Implementación

1. **Entrada:** Leer las dimensiones $H \times W$ de la imagen y sus $H \times W$ valores enteros de intensidad.
2. **Imagen integral:** Calcular, para cada posición (i, j) (indexación desde 0, `[fila][columna]`),

$$II(i,j) = \sum_{i' \le i,\ j' \le j} I(i', j'),$$

   es decir, la suma de todos los píxeles arriba y a la izquierda de (i, j), incluyendo la propia posición.
3. **Consultas:** Leer el entero Q y, a continuación, Q líneas, cada una con cuatro enteros x_1 y_1 x_2 y_2 — las esquinas superior-izquierda e inferior-derecha de un rectángulo, **ambas inclusivas**, con $0 \le x_1 \le x_2 < W$ y $0 \le y_1 \le y_2 < H$.
4. **Suma rectangular en O(1):** Para cada consulta, calcular la suma de intensidades dentro del rectángulo usando exclusivamente valores ya presentes en II (sin recorrer los píxeles originales):

$$S(x_1, y_1, x_2, y_2) = II(y_2, x_2) - II(y_2, x_1{-}1) - II(y_1{-}1, x_2) + II(y_1{-}1, x_1{-}1),$$

   tratando cualquier término con índice de fila o columna igual a -1 como 0.
5. **Salida:** Primero, imprimir la imagen integral completa — H líneas con W enteros cada una. Luego, para cada consulta, imprimir un único entero: la suma de la región correspondiente.

8.14.3.2 📌 Restricciones Computacionales

- **No recalcular por fuerza bruta:** la respuesta a cada consulta debe usar la fórmula de cuatro términos sobre II, no una suma directa de los píxeles del rectángulo (aunque el resultado numérico sea el mismo, el objetivo del ejercicio es justamente esa técnica).
- **Rectángulos con coordenadas inclusivas:** (x_1, y_1) y (x_2, y_2) pertenecen a la región sumada.
- **Tratamiento de borde:** al consultar II con índice -1 (cuando $x_1 = 0$ o $y_1 = 0$), utilizar el valor 0.

8.14.3.3 🧠 Fundamentación Teórica

Elemento	Papel en el Haar Cascade
Imagen integral II	Precomputada una única vez por imagen, en tiempo $O(HW)$
Consulta en O(1)	Cada característica Haar (diferencia entre sumas de regiones rectangulares) se evalúa con pocas operaciones, independientemente del área del rectángulo
Escalabilidad	Es esa constancia la que hace viable evaluar miles de características, en múltiples posiciones y escalas, en tiempo real
Principio de inclusión-exclusión	Los cuatro términos de la fórmula suman la región deseada y restan exactamente las áreas contadas en exceso

8.14.3.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

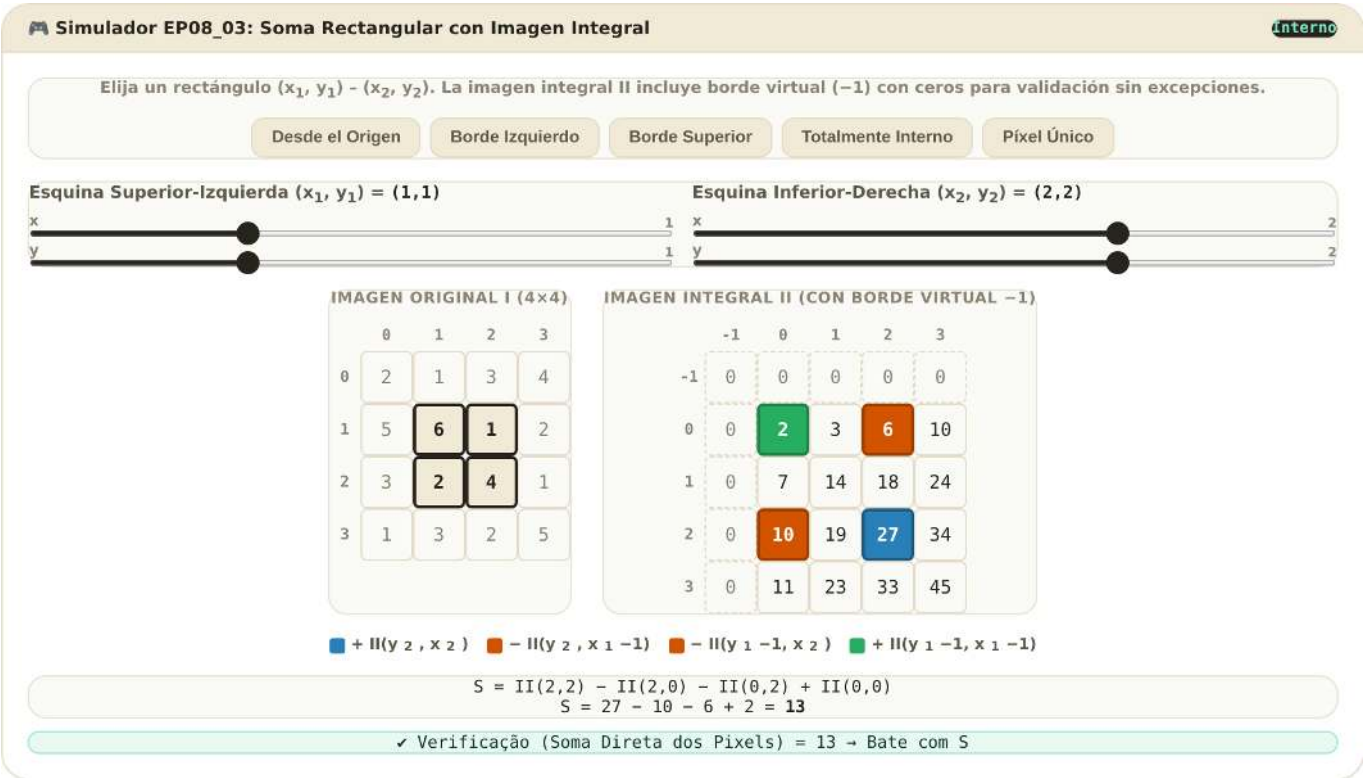
- Línea 1: Enteros H y W.
- Siguientes H líneas: W enteros cada una (imagen original).
- Siguiente línea: Entero Q.
- Siguientes Q líneas: cuatro enteros x_1 y_1 x_2 y_2.

Salida:

- H líneas con W enteros cada una (la imagen integral).
- Q líneas, una por consulta, con la suma de la región correspondiente.

8.14.3.5 📌 Ejemplos

Entrada	Salida	Observación
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	La consulta cubre la imagen completa; la suma coincide con $II(2,2)$ y con la suma de todos los 9 valores.
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	La primera consulta usa los cuatro términos de la fórmula; la segunda coincide directamente con $II(1,1)$, pues comienza en el origen.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap08/sim-ep0803.html

Figura 8.17: Simulador EP08_03: Soma Rectangular en O(1) — Múltiples Situaciones de Borde

```
%%writefile EP08_03.py
# Código Python
```

```
Overwriting EP08_03.py
```

```
TestSuite("EP08_03.py").run()
```

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

8.14.4 EP08_04 🟢 IoU y Supresión de No Máximos (NMS)

La figura de esta sección mostró el efecto de la Supresión de No Máximos sobre un conjunto de cajas producidas por un detector tipo *sliding window*: múltiples detecciones redundantes por objeto se redujeron a una única caja por objeto. Se le ha encargado reimplementar, byte a byte, las dos funciones que produjeron ese resultado — `calcular_iou` y `supresion_no_maximos` — para confirmar, con sus propias manos, exactamente los números que el capítulo presentó.

8.14.4.1 📋 Directrices de Implementación

1. **Entrada:** Leer el entero N (número de cajas) y el real τ (umbral de IoU). A continuación, leer N líneas, cada una con cinco reales x_{min} y_{min} x_{max} y_{max} score.

2. **Intersección sobre Unión:** Para dos cajas A y B,

$$\text{IoU}(A, B) = \frac{\text{área}(A \cap B)}{\text{área}(A \cup B)},$$

   con área de intersección nula cuando las cajas no se superponen.

3. **Algoritmo de NMS** (exactamente como se describe en el capítulo):

a. Ordenar las cajas por `score` descendente (los empates mantienen el orden de lectura original).

b. Seleccionar la caja de mayor puntuación entre las restantes; añadirla a la salida y eliminarla de la lista.

c. Descartar, de la lista restante, **todas** las cajas cuyo IoU con la caja seleccionada sea **mayor o igual** a τ — solo las cajas con IoU $< \tau$ permanecen como candidatas.

d. Repetir (b)–(c) hasta que la lista de restantes esté vacía.

4. **Salida:** Para cada caja mantenida, en el orden en que fue seleccionada, imprimir su índice original (posición de lectura, a partir de 0) y su `score`, con 2 decimales. Al final, imprimir `Total mantenidas: X`.

8.14.4.2 📌 Restricciones Computacionales

- **Atención al sentido del umbral:** al contrario de lo que se podría suponer, una caja es **suprimida** cuando IoU $\geq \tau$ (no solo cuando IoU $> \tau$) — seguir exactamente ese criterio, el mismo del código de referencia del capítulo.
- **Índices originales:** la salida hace referencia a la posición de lectura de cada caja en la entrada, no a su posición después de la ordenación por `score`.
- **Área sin suma de 1 píxel:** usar área $= (x_{max} - x_{min}) \times (y_{max} - y_{min})$, exactamente como en el capítulo (sin el ajuste "+1" a veces usado en otras convenciones).

8.14.4.3 🧠 Fundamentación Teórica

Elemento	Papel en el postprocesamiento
IoU	Cuantifica la superposición espacial entre dos cajas delimitadoras
Sliding window (Haar Cascade)	Produce típicamente varias detecciones superpuestas para el mismo objeto, en posiciones y escalas cercanas
Umbral τ	Controla la agresividad de la supresión: demasiado bajo fusiona objetos cercanos; demasiado alto deja pasar redundancias
Ordenación por `score`	Garantiza que, entre cajas redundantes, la de mayor confianza siempre sobrevive

8.14.4.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

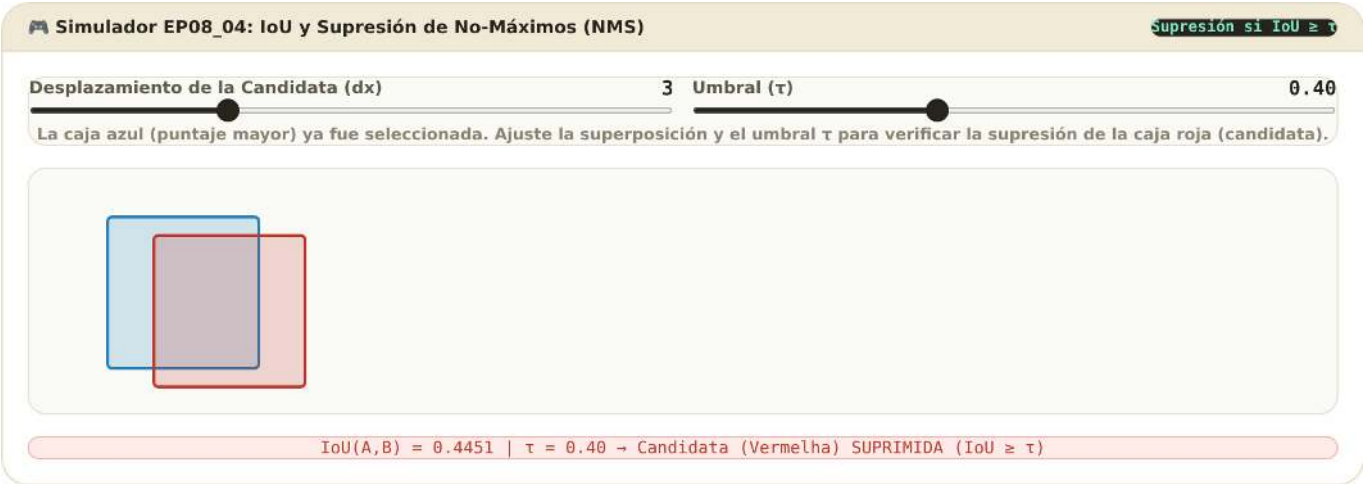
- Línea 1: Entero N y real τ.
- Siguientes N líneas: cinco reales x_{min} y_{min} x_{max} y_{max} score.

Salida:

- Una línea por caja mantenida, en el orden de selección: `índice score` (score con 2 decimales).
- Última línea: `Total mantenidas: X`.

8.14.4.5 📌 Ejemplos

Entrada	Salida	Observación
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 mantenidas: 2	Exactamente el ejemplo de la figura del capítulo: 5 cajas redundantes (2 objetos) se convierten en 2 detecciones finales. El IoU entre la 1.ª y la 2.ª cajas es $\approx 0{,}775$, muy por encima de $\tau = 0{,}4$.



Versión interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap08/sim-ep0804.html

Figura 8.18: Simulador EP08_04: IoU y Supresión de No-Máximos

```
%%writefile EP08_04.py
# Código Python
```

```
Overwriting EP08_04.py
```

```
TestSuite("EP08_04.py").run()
```

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

8.14.5 EP08_05 🟡 Etiquetado de Componentes Conectados: Segmentación de Instancias

El ejemplo de segmentación clásica de este capítulo separó "instancias" de monedas simplemente por su desconexión espacial en la máscara binaria resultante de la umbralización de Otsu. Esa etapa final — etiquetar cada componente conectado con un identificador de instancia — es exactamente lo que se te ha encargado implementar aquí, desde cero, sobre una máscara binaria ya preparada (0 = fondo, 1 = objeto), como si fuera una reimplementación manual de `cv2.connectedComponents`.

Este ejercicio también expone, de forma muy concreta, la limitación discutida en el capítulo: el resultado depende enteramente de cómo se define "vecindad" entre píxeles — y, como verás en el segundo ejemplo, dos píxeles en diagonal pueden considerarse la misma instancia o instancias diferentes, dependiendo exclusivamente de la **conectividad** elegida, no de ninguna noción semántica de objeto.

8.14.5.1 📋 Directrices de Implementación

1. **Entrada:** Leer las dimensiones $H \times W$ de la máscara binaria y sus $H \times W$ valores (0 o 1).

2. **Conectividad:** Leer el entero $c \in \{4, 8\}$. En la conectividad 4, los vecinos de (i, j) son $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ y $(i, j+1)$. En la conectividad 8, se suman las cuatro diagonales: $(i-1, j-1)$, $(i-1, j+1)$, $(i+1, j-1)$ y $(i+1, j+1)$.
3. **Descubrimiento de componentes:** Recorriendo la máscara en un barrido línea a línea, de izquierda a derecha y de arriba hacia abajo, siempre que se encuentre un píxel de valor 1 aún sin etiqueta, este inicia un **nuevo componente**: asígnale la siguiente etiqueta disponible (el primer componente descubierto recibe la etiqueta 1, el segundo la etiqueta 2, y así sucesivamente) y propaga esa misma etiqueta a todos los píxeles de valor 1 alcanzables desde él mediante una cadena de vecinos (de acuerdo con la conectividad elegida) — por búsqueda en anchura, en profundidad, o *union-find*, a tu elección.
4. **Píxeles de fondo:** permanecen con etiqueta 0 y no pertenecen a ninguna instancia.
5. **Salida:** Primero, imprimir el mapa de etiquetas completo — H líneas con W enteros cada una. A continuación, para cada etiqueta ℓ de 1 a K (en el orden de descubrimiento), imprimir `Instancia l: A píxeles`, donde A es la cantidad de píxeles con esa etiqueta. Por último, imprimir `Total de instancias: K`.

8.14.5.2 📌 Restricciones Computacionales

- **Orden de descubrimiento = orden de barrido:** las etiquetas se numeran en el orden en que cada nuevo componente se encuentra mediante el barrido línea a línea, no por tamaño ni posición.
- **Conectividad explícita:** dos píxeles de valor 1 solo pertenecen a la misma instancia si existe una cadena de vecinos **de acuerdo con** c que los conecte entre sí — no uses la conectividad opuesta por error.
- **Máscara binaria pura:** todos los valores de entrada son exactamente 0 o 1.

8.14.5.3 🧠 Fundamentación Teórica

Elemento	Papel en la segmentación clásica de instancias
Umbralización (Otsu, Cap. 4)	Etapa anterior que produce la máscara binaria a partir de la imagen de intensidad
Componente conectado	Cada instancia se define **solo** por la conectividad espacial de los píxeles de objeto, sin ninguna noción de forma, clase o apariencia
Conectividad 4 vs. 8	Parámetro que altera el resultado: bajo conectividad 8, dos blobs unidos solo en diagonal se convierten en una única instancia
Limitación central	La técnica fusiona instancias que se tocan o se superponen (aunque sean objetos claramente distintos), pues no hay noción de "objeto" — solo de "región conectada"

8.14.5.4 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Enteros H y W.
- Siguientes H líneas: W enteros (0 o 1) cada una.
- Última línea: Entero c (4 o 8).

Salida:

- H líneas con W enteros cada una (el mapa de etiquetas).
- Una línea por instancia, en el orden de descubrimiento: `Instancia l: A píxeles`.
- Última línea: `Total de instancias: K`.

8.14.5.5 📌 Ejemplos

Entrada	Salida	Observación
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 Instancia 1: 4 píxeles Instancia 2: 4 píxeles Total de instancias: 2	Dos bloques 2×2 claramente separados: el resultado es el mismo bajo conectividad 4 u 8.
2 2 1 0 0 1 8	1 0 0 1 Instancia 1: 2 píxeles Total de instancias: 1	Bajo conectividad 8, los dos píxeles en diagonal pertenecen a la **misma** instancia. Repite este ejemplo con $c = 4$: el resultado pasa a ser 2 instancias de 1 píxel cada una — puramente por el cambio de conectividad, sin ninguna diferencia en la máscara.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap08/sim-ep0805.html

Figura 8.19: Simulador EP08_05: Etiquetado de Componentes Conectados — Conectividad 4 vs. 8

```
%%writefile EP08_05.py
# Código Python
```

```
Overwriting EP08_05.py
```

```
TestSuite("EP08_05.py").run()
```

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

8.14.6 EP08_06 🟡 *Bounding Boxes*, Centroides y Propiedades de Instancias con `mm.measure`

En el ejercicio anterior (**EP08_05**), se puede observar cómo la segmentación por componentes conectados etiqueta regiones binarias contiguas para separar instancias. Sin embargo, para tareas de detección, seguimiento

y análisis cuantitativo de objetos, el simple mapa de etiquetas no es suficiente. Se vuelve necesario extraer **métricas espaciales y geométricas** que caractericen cada instancia individualmente.

Este EP se centra en el cálculo y la extracción automática de las propiedades fundamentales de visión por computadora para cada componente conectado encontrado en la máscara binaria, utilizando el método nativo `mm.measure(img)` de la biblioteca `morph`:

1. **Caja Delimitadora (*Bounding Box*):** El rectángulo más pequeño alineado con los ejes que envuelve completamente la instancia, definido por su esquina superior izquierda (x, y), ancho w y alto h.
2. **Centroide Geométrico** $(\bar{x}, \bar{y})$**:** El centro de masa de la instancia en la cuadrícula discreta, equivalente a los momentos espaciales de primer orden M_{10}/M_{00} y M_{01}/M_{00}.
3. **Área Geométrica del Contorno** (A)**:** El área delimitada por el contorno de la instancia calculada mediante `mm.contourArea(c)`.

8.14.6.1 📋 Directrices de Implementación

1. **Entrada:** Leer las dimensiones $H \times W$ de la máscara binaria, los $H \times W$ valores (0 o 1) y el parámetro de conectividad $c \in \{4, 8\}$.
2. **Extracción Automática con `mm.measure`:** Pasar la imagen binarizada a la función `mm.measure(img_bin)`, que extrae los contornos OpenCV y devuelve una lista de diccionarios que contienen las propiedades geométricas de cada instancia.
3. **Propiedades Devueltas:** Para cada diccionario m de la lista devuelta por `medidas = mm.measure(img_bin)`:
   - **Área (`area`):** Valor numérico del área geométrica del contorno `mm.contourArea(c)`.
   - **Bounding Box (`bbox`):** Tupla (x, y, w, h) que representa la esquina superior izquierda, el ancho y el alto.
   - **Centroide (`center`):** Tupla (c_x, c_y) con las coordenadas del centro de masa M_{10}/M_{00} y M_{01}/M_{00}. Formatear con **dos decimales**.
4. **Salida:** Para cada instancia $1, \ldots, K$ encontrada (ordenada por orden de descubrimiento/posición en la imagen), imprimir una línea que contenga sus propiedades. Finalmente, imprimir el número total de instancias.
   - Para ordenar, usar `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`.

8.14.6.2 🧠 Fundamentación Teórica

Propiedad en `mm.measure`	Cálculo Matemático / Lógica Discreta	Aplicación Práctica en Visión
`bbox` (OpenCV)	$[x, y, w, h] = [\min(c), \min(r), \Delta c + 1, \Delta r + 1]$	Formato clásico de OpenCV. *Nota: redes como YOLO convierten este rectángulo a (c_x, c_y, w, h) normalizado.*
`center`	$\bar{x} = \frac{M_{10}}{M_{00}}, \quad \bar{y} = \frac{M_{01}}{M_{00}}$	Centro de masa exacto de la máscara (usado en seguimiento y análisis de trayectoria).
`area`	$A = \text{contourArea}(C)$ (Fórmula del Polígono)	Métrica continua de la superficie del objeto.

8.14.6.3 📦 Especificación de Entrada y Salida (VPL)

Entrada:

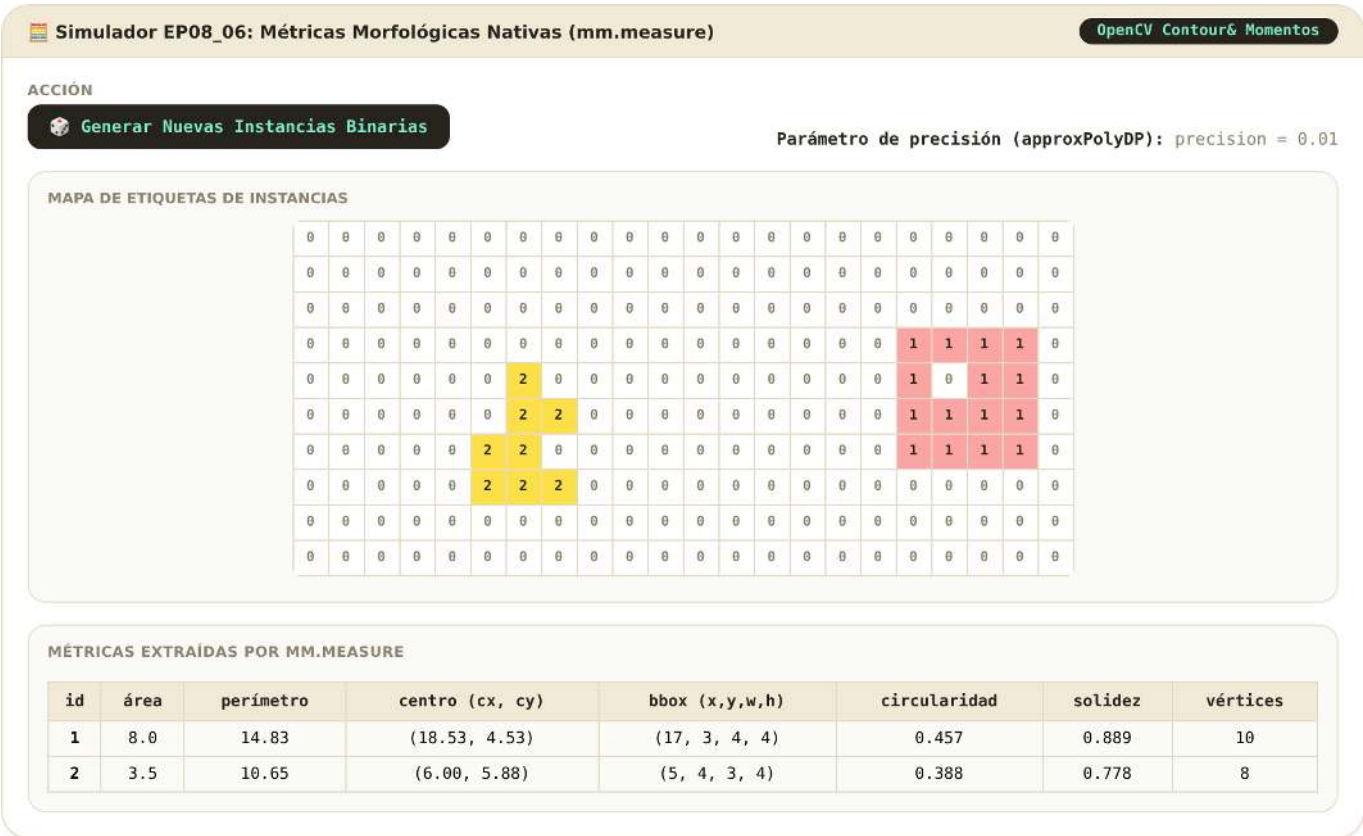
- Línea 1: Enteros H y W.
- Siguientes H líneas: W enteros (0 o 1) cada una.
- Última línea: Entero c (4 o 8).

Salida:

- Una línea por instancia en el orden de descubrimiento: `Instancia l: Area=A, BBox=(x,y,w,h), Centroide=(cx,cy)`
- Última línea: `Total de instancias: K`.

8.14.6.4 📌 Ejemplos

Entrada	Salida	Observación
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	Instancia 1: Area=1.0, BBox=(1,1,2,2), Centroide=(1.50,1.50) Instancia 2: Area=1.0, BBox=(4,4,2,2), Centroide=(4.50,4.50) Total de instancias: 2	Bloques 2×2 alineados. El cálculo del área geométrica del contorno resulta en 1.0. El centroide del bloque en las columnas 1–2 y filas 1–2 es exactamente (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	Instancia 1: Area=2.0, BBox=(1,1,4,2), Centroide=(2.40,1.20) Total de instancias: 1	Objeto asimétrico en forma de "T" invertida. El área geométrica del contorno es 2.0. El centroide refleja la distribución de los píxeles del objeto.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap08/sim-ep0806.html

Figura 8.20: Simulador EP08_06: Extracción de *Bounding Boxes*, Centroides y Propiedades con *mm.measure*

```
%%writefile EP08_06.py
# Código Python
```

```
Overwriting EP08_06.py
```

```
TestSuite("EP08_06.py").run()
```

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

8.14.7 EP08_07 🟡 Eliminación de Ruido Sal y Pimienta y Medición de Objetos

En este ejercicio, usted aplicará filtrado morfológico para limpiar una imagen binaria corrompida por ruido del tipo **sal y pimienta** (píxeles aislados de valor `1` en el fondo y `0` en el interior de los objetos). Después de la limpieza, el programa debe extraer las mediciones geométricas de los componentes conectados restantes, ordenarlos y mostrar la tabla final de métricas.

8.14.7.1 📋 Directrices de Implementación

1. **Entrada:** leer dos enteros H y W (alto y ancho de la imagen) en la primera línea y, a continuación, las H líneas con la matriz binaria que contiene píxeles `0` y `1` separados por espacio.
2. **Filtrado Morfológico:** aplicar encadenamiento de **Apertura** (para eliminar el ruido sal en el fondo) seguido de **Cierre** (para rellenar el ruido pimienta dentro de los objetos) con elemento estructurante 3×3.
3. **Impresión de la Imagen Limpia:** imprimir la matriz resultante en valores `0` y `1` separados por espacio.
4. **Mediciones Geométricas:** para cada objeto identificado en la matriz limpia, extraer:

- `id`: identificador numérico secuencial (reasignado después de la ordenación);
- `area`: área calculada mediante contorno (`cv2.contourArea`);
- `perimeter`: perímetro del contorno (`cv2.arcLength`);
- `cx`, `cy`: centro de masa (centroide mediante `cv2.moments`);
- `x`, `y`, `w`, `h`: coordenadas del rectángulo delimitador (`cv2.boundingRect`);
- `circularity`: circularidad dada por $\frac{4\pi \cdot \text{área}}{\text{perímetro}^2}$;
- `solidity`: solidez dada por la razón $\frac{\text{área}}{\text{área del casco convexo}}$;
- `vertices`: número de vértices aproximado del polígono (`cv2.approxPolyDP` con $\epsilon = 0.02 \times$ perímetro).

5. **Ordenación y Salida:** ordenar los objetos en orden creciente por la posición X del rectángulo delimitador (`bbox[0]`); en caso de empate, usar la posición Y (`bbox[1]`). Reasignar los `ids` de 1 a N e imprimir la tabla formateada.
   - Para ordenar, usar `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, con `medidas = mm.measure(img)`.

8.14.7.2 📌 Restricciones y Reglas de Ordenación

- **Regla de Ordenación de los Objetos:**

```
medidas.sort(key=lambda m: (m['bbox'][0], m['bbox'][1]))
```

- **Diferencia de Área:** El área calculada por OpenCV (`cv2.contourArea`) mide el área del polígono continuo delimitado por los centros de los píxeles de borde, resultando en valores numéricos menores que el simple conteo discreto de píxeles `1` (`np.sum`).

8.14.7.3 🧠 Fundamentación Teórica

Operación / Métrica	Función en el Filtrado y Caracterización
Apertura Morfológica ($\circ$)	Erosión seguida de dilatación: elimina ruidos brillantes aislados (*sal*).
Cierre Morfológico ($\bullet$)	Dilatación seguida de erosión: rellena pequeños huecos oscuros en el interior de los objetos (*pimienta*).
`cv2.boundingRect`	Devuelve (x, y, w, h), el rectángulo más pequeño alineado a los ejes que envuelve al objeto.
Circularidad y Solidez	Describen la compacidad y la convexidad geométrica del componente.

8.14.7.4 📌 Ejemplos

Entrada	Salida
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
# Código Python
```

```
Overwriting EP08_07.py
```

```
TestSuite("EP08_07.py").run()
```

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

8.14.8 EP08_08 🟡 Imagen en Niveles de Gris y Umbralización Dinámica

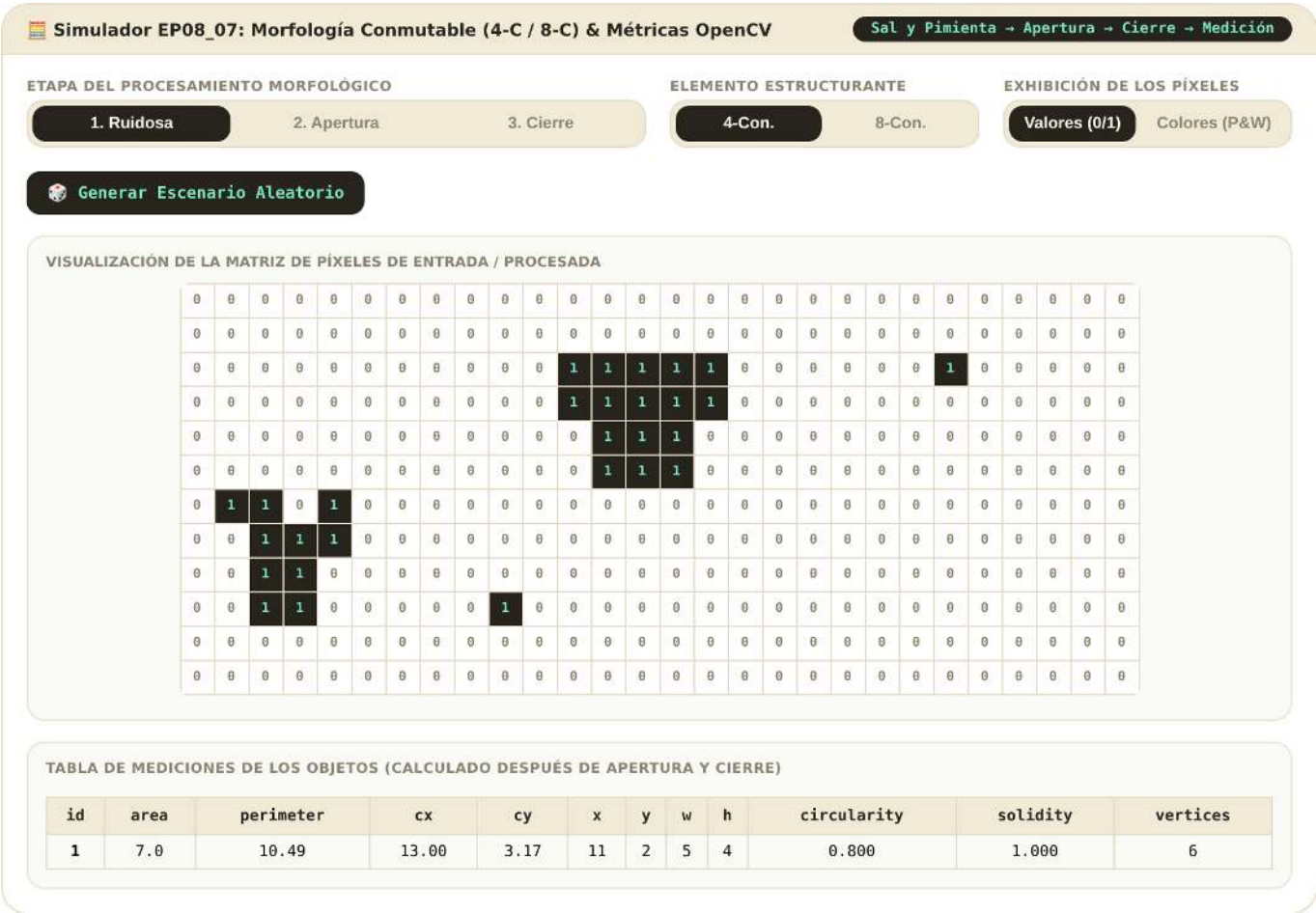
En este ejercicio, la imagen de entrada deja de ser estrictamente binaria (`0/1`) y pasa a ser una **imagen en niveles de gris** (8 **bits,** $0 \ldots 255$), donde los objetos poseen una intensidad media intermedia sobre un fondo oscuro (0), además de ruido de tipo sal y pimienta distribuido por toda la imagen.

8.14.8.1 📋 Directrices de Implementación

1. **Entrada:** leer H y W en la primera línea, seguidos de las H líneas con valores enteros de 0 a 255 en una matriz de $H \times W$.
2. **Preprocesamiento:**

- Aplicar un filtro de **Mediana** (3×3) para eliminar el ruido de sal y pimienta manteniendo los bordes nítidos.
- Aplicar **Umbralización de Otsu** (o un umbral fijo $T = 60$) para binarizar la imagen limpia.

3. **Medición y Salida:** extraer el contorno de los objetos, calcular las métricas geométricas (`area`, `perimeter`, `cx`, `cy`, `x`, `y`, `w`, `h`, `circularity`, `solidity`) y ordenar los objetos por `bbox[0]` (y `bbox[1]` como criterio de desempate). Reasignar `id` de 1 a N e imprimir la tabla.
   - Para ordenar, usar `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, con `medidas = mm.measure(img)`.



id	area	perimeter	cx	cy	x	y	w	h	circularity	solidity	vertices
1	7.0	10.49	13.00	3.17	11	2	5	4	0.800	1.000	6

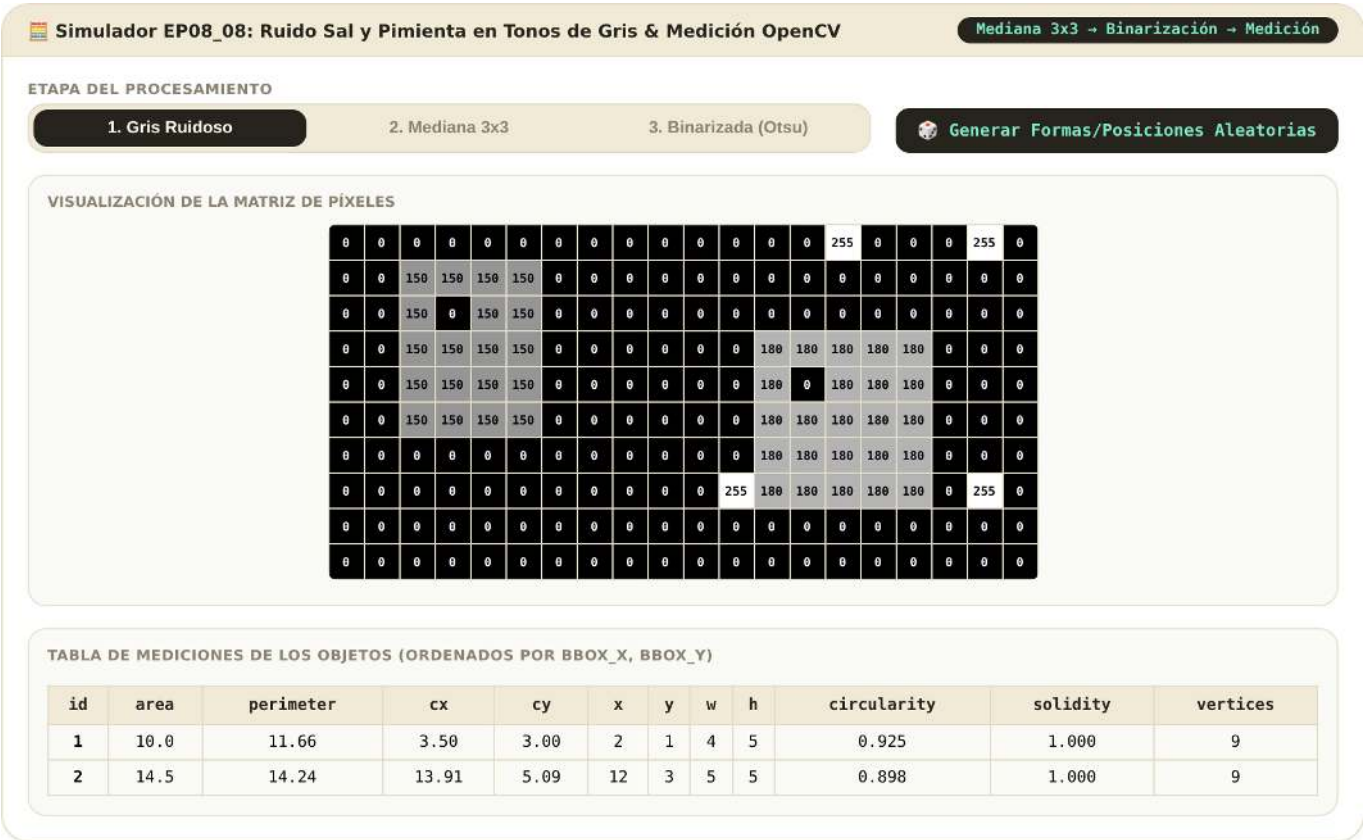
Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap08/sim-ep0807.html

Figura 8.21: Simulador EP08_07: Morfología con Conectividad Configurable y Medición

8.14.8.2 📌 Ejemplos

Entrada	Salida
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 … (imagen binaria que contiene un cuadrado y un círculo)	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

[req 961564] Traduce este Markdown de portugués a español:



id	area	perimeter	cx	cy	x	y	w	h	circularity	solidity	vertices
1	10.0	11.66	3.50	3.00	2	1	4	5	0.925	1.000	9
2	14.5	14.24	13.91	5.09	12	3	5	5	0.898	1.000	9

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap08/sim-ep0808.html

Figura 8.22: Simulador EP08_08: Filtrado de Mediana en Tonos de Gris y Medición de Objetos OpenCV

```
%%writefile EP08_08.py
# Código Python
```

```
Overwriting EP08_08.py
```

```
TestSuite("EP08_08.py").run()
```

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

8.14.9 EP08_09 🟠 Gradiente de Iluminación y Umbralización Adaptativa

En esta variación, los objetos están inmersos en un fondo con **iluminación no uniforme (gradiente suave de iluminación)**. La umbralización simple por valor único falla, lo que exige un preprocesamiento más robusto.

8.14.9.1 📋 Directrices de Implementación

1. **Entrada:** imagen en niveles de gris $H \times W$ con variación de fondo de 20 a 180.
2. **Preprocesamiento:**

- Aplicar **Umbralización Adaptativa** (ej.: `cv2.adaptiveThreshold` con ventana gaussiana de 15×15 y constante $C = 3$) para aislar los objetos independientemente de la variación del fondo.

  `cv2.adaptiveThreshold(img_gray, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, ksize, C) // 255`

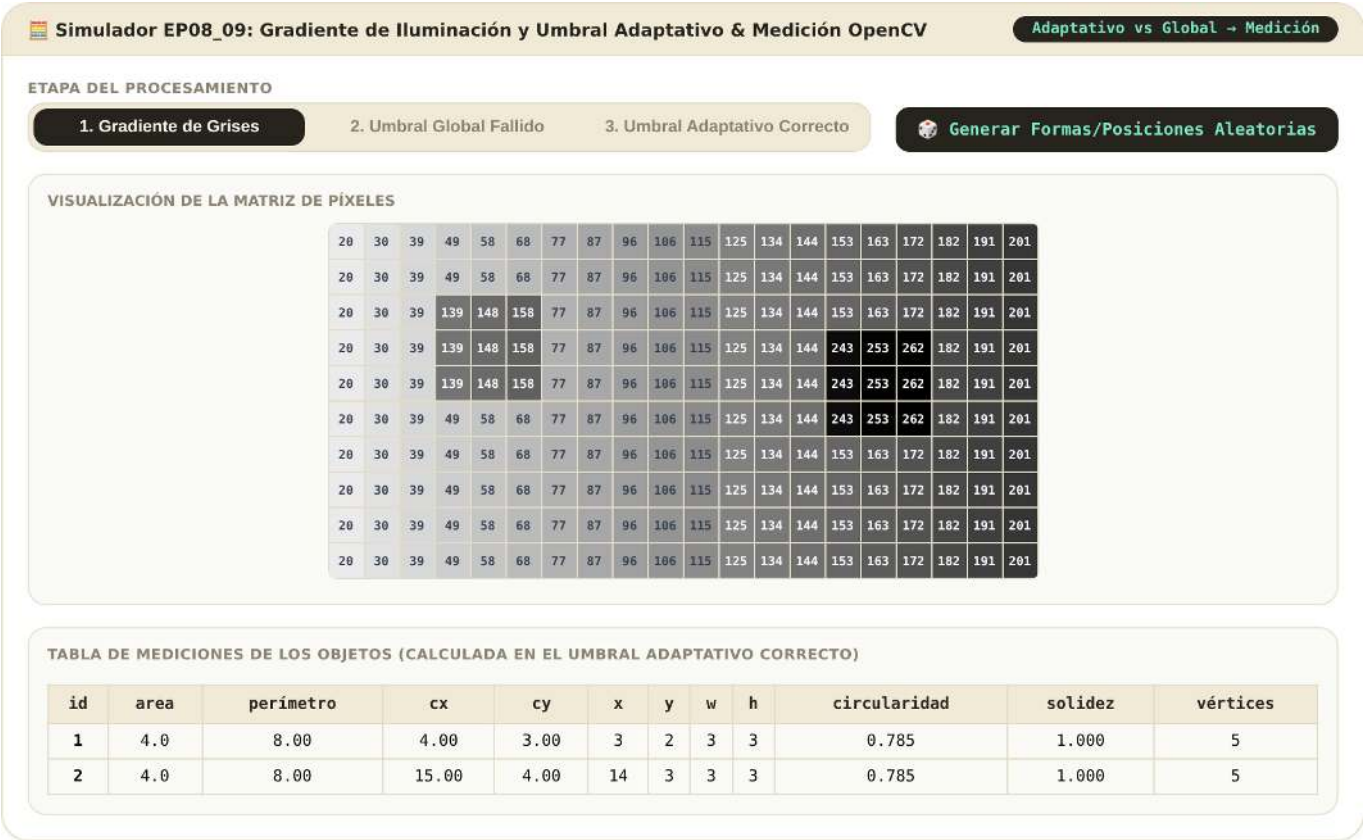
  `ksize` y `C` se leen después de la imagen.

- Operación morfológica de **Cierre** (3×3) para sellar posibles fallos en los contornos.

1. **Medición y Clasificación:** extraer las medidas.
2. **Ordenación y Salida:** ordenar por `(bbox[0], bbox[1])` e imprimir la tabla incluyendo la columna `class`.
   - Para ordenar, usar `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, con `medidas = mm.measure(img, precision=0.02)`.

8.14.9.2 📌 Ejemplos

Entrada	Salida
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 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.es/cap08/sim-ep0809.html

Figura 8.23: Simulador EP08_09: Gradiente de Iluminação, Limiar Adaptativo y Medición OpenCV

```
%%writefile EP08_09.py
# Código Python
```

```
Overwriting EP08_09.py
```

```
TestSuite("EP08_09.py").run()
```

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

8.14.10 EP08_10 🔴 Contraste Bajo y Separación de Objetos Tópicos (*Watershed* / Distancia)

En este ejercicio, **algunos objetos geométricos están ligeramente en contacto (superpuestos en los bordes)**. La simple extracción de contornos trataría dos objetos como uno solo.

8.14.10.1 📋 Directrices de Implementación

1. **Entrada:** matriz $H \times W$ en niveles de gris con objetos de intensidad $110 \ldots 140$ sobre fondo 0, con ruido y pares de objetos tangentes.
2. **Preprocesamiento y Separación:**

- Aplicación de la umbralización.
- Aplicación de la **Transformada de Distancia** (`mm.dist`).
- Obtención de los picos de distancia para que sirvan como marcadores en la **Transformada *Watershed*** (`mm.watershed`), separando físicamente los objetos en contacto en la máscara. **Consejo:** usar `mm.regmax()` para obtener los máximos locales y luego etiquetar con `mm.label0`.
- Después del *watershed*, aplicar nuevamente la umbralización con `mm.threshold(water,0)//255`.

3. **Análisis de Componentes Conectados:** medir cada región aislada después del *Watershed*.
4. **Salida:** imprimir los componentes ordenados por `(bbox[0], bbox[1])` con sus métricas individuales de área, centroide y solidez.
   - Para ordenar, usar `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, con `medidas = mm.measure(img, precision=0.02)`.

8.14.10.2 📌 Ejemplos

Entrada	Salida
16 16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 … (imagen binaria que contiene dos cuadrados)	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

[req 961564] Traduce este Markdown de portugués a español:

Simulador EP08_10: Separación de Discos Tangentes (Transformada L2 & Watershed)

mm.dist L2 → mm.watershed → Medición

ETAPA DEL PROCESAMIENTO MORFOLÓGICO

1. Gris Ruidoso | 2. Máscara Unida | 3. Distancia L2 | 4. Watershed (Corte)

Generar Discos con Radios Aleatorios

VISUALIZACIÓN DE LA MATRIZ DE PÍXELES

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	150	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	154	146	149	150	156	0	0	0	0	0	0	0	0	0
0	0	0	158	0	0	149	142	146	159	151	157	147	0	0	0	0	0	0	0	0
0	0	145	156	156	0	155	158	159	144	155	154	147	0	0	0	0	0	0	0	0
0	146	146	153	151	149	149	141	158	156	158	158	154	140	0	0	0	0	0	0	0
0	0	153	157	147	0	147	152	153	155	157	146	145	0	0	0	0	0	0	0	0
0	0	0	145	0	0	156	150	157	146	151	157	155	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	143	142	156	143	148	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	147	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

TABLA DE MEDICIONES DE LOS DISCOS DESPUÉS DEL CORTE DEL WATERSHED

id	área	perímetro	cx	cy	x	y	w	h	circularidad	solidez	vértices
1	7.0	10.49	2.83	5.00	1	3	4	5	0.800	1.000	6
2	37.0	24.14	9.08	5.00	6	1	8	9	0.798	0.925	9

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap08/sim-ep0810.html

Figura 8.24: Simulador EP08_10: Separación de Discos Tangentes mediante la Transformada de Distancia L2 y *Watershed*

```
%%writefile EP08_10.py
# Código Python
```

```
Overwriting EP08_10.py
```

```
TestSuite("EP08_10.py").run()
```

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

8.14.11 EP08_11 🔴 Clasificación y Validación de Objetos con Plantilla de *Bounding Box*

En este ejercicio, el objetivo es procesar una imagen en tonos de gris que contiene múltiples objetos geométricos, extraer sus propiedades con `mm.measure` y validar las cajas delimitadoras (*bounding boxes*) detectadas en relación con una plantilla real (*Ground Truth* - GT) proporcionada en la entrada, utilizando la métrica IoU (*Intersection over Union*).

8.14.11.1 📋 Directrices de Implementación

1. **Lectura de la Imagen:** Leer las dimensiones $H \times W$ y la matriz $H \times W$ de píxeles de la imagen en tonos de gris.
2. ***Pipeline* Morfológico:** Binarizar la imagen mediante el método de Otsu (`mm.threshold`) y renderizar la máscara binarizada resultante utilizando `mm.drawImg`.
3. **Lectura de la Plantilla Real (*Ground Truth*):**

- Leer la cantidad G de cajas delimitadoras de la plantilla.
- Si $G > 0$, leer G líneas que contienen 5 valores cada una: `id xmin_norm ymin_norm xmax_norm ymax_norm`.
- **Conversión de Coordenadas:** Las coordenadas de la plantilla están normalizadas en el rango $[0.0, 1.0]$. Para convertir a píxeles en la cuadrícula de la imagen:

$$x_{\min} = \lfloor \text{xmin_norm} \times W \rfloor, \quad y_{\min} = \lfloor \text{ymin_norm} \times H \rfloor$$

$$w = \lfloor \text{xmax_norm} \times W \rfloor - x_{\min}, \quad h = \lfloor \text{ymax_norm} \times H \rfloor - y_{\min}$$

4. **Extracción de Métricas y Cálculo de IoU:**

- Extraer las propiedades de las instancias con `mm.measure(img_bin, precision=0.02)`.
- Para cada *bounding box* detectada (x, y, w, h), calcular la superposición IoU respecto a las cajas de la plantilla y definir `hits = 1` si existe alguna coincidencia con IoU ≥ 0.50, o `hits = 0` en caso contrario.

5. **Salida:** Ordenar las instancias por posición `(bbox[0], bbox[1])` e imprimir la tabla CSV con la columna adicional `hits`.
   - Para ordenar, usar `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, con `medidas = mm.measure(img)`.

8.14.11.2 🧠 Fundamentación Teórica y Conversión

Concepto	Fórmula / Operación	Descripción
BBox Detectada	(x, y, w, h) vía `mm.measure`	Caja delimitadora calculada en la cuadrícula discreta en píxeles enteros.

Concepto	Fórmula / Operación	Descripción
BBox Plantilla (GT)	$(x_{\min}, y_{\min}, w, h)$ convertidos	Caja real proporcionada en la entrada en coordenadas relativas $[0.0, 1.0]$.
IoU (Intersection over Union)	$\text{IoU} = \frac{\text{Área}(B_{\text{DET}} \cap B_{\text{GT}})}{\text{Área}(B_{\text{DET}} \cup B_{\text{GT}})}$	Evalúa la tasa de superposición de las cajas. Se considera válida si $\text{IoU} \geq 0.50$.
Estado de Validación (`hits`)	1 si $\max(\text{IoU}) \geq 0.50$, sino 0	Indicador binario de acierto del detector respecto a la plantilla.

8.14.11.3 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- **Línea 1:** Enteros H y W (dimensiones de la matriz).
- **Siguientes H líneas:** W enteros (0 a 255) que representan la imagen en tonos de gris.
- **Línea $H+2$:** Entero G (cantidad de cajas de la plantilla verdadera).
- **Siguientes G líneas:** 5 valores numéricos por línea: `id xmin_norm ymin_norm xmax_norm ymax_norm` (donde las coordenadas son valores flotantes entre 0.0 y 1.0).

Salida:

1. Matriz binarizada renderizada vía `mm.drawImg(img_bin)`.
2. Encabezado CSV: `id,area,perimeter,cx,cy,x,y,w,h,circularity,solidity,vertices,hits`
3. Una línea CSV por objeto detectado que contiene sus propiedades formateadas y el indicador `hits` (1 o 0).

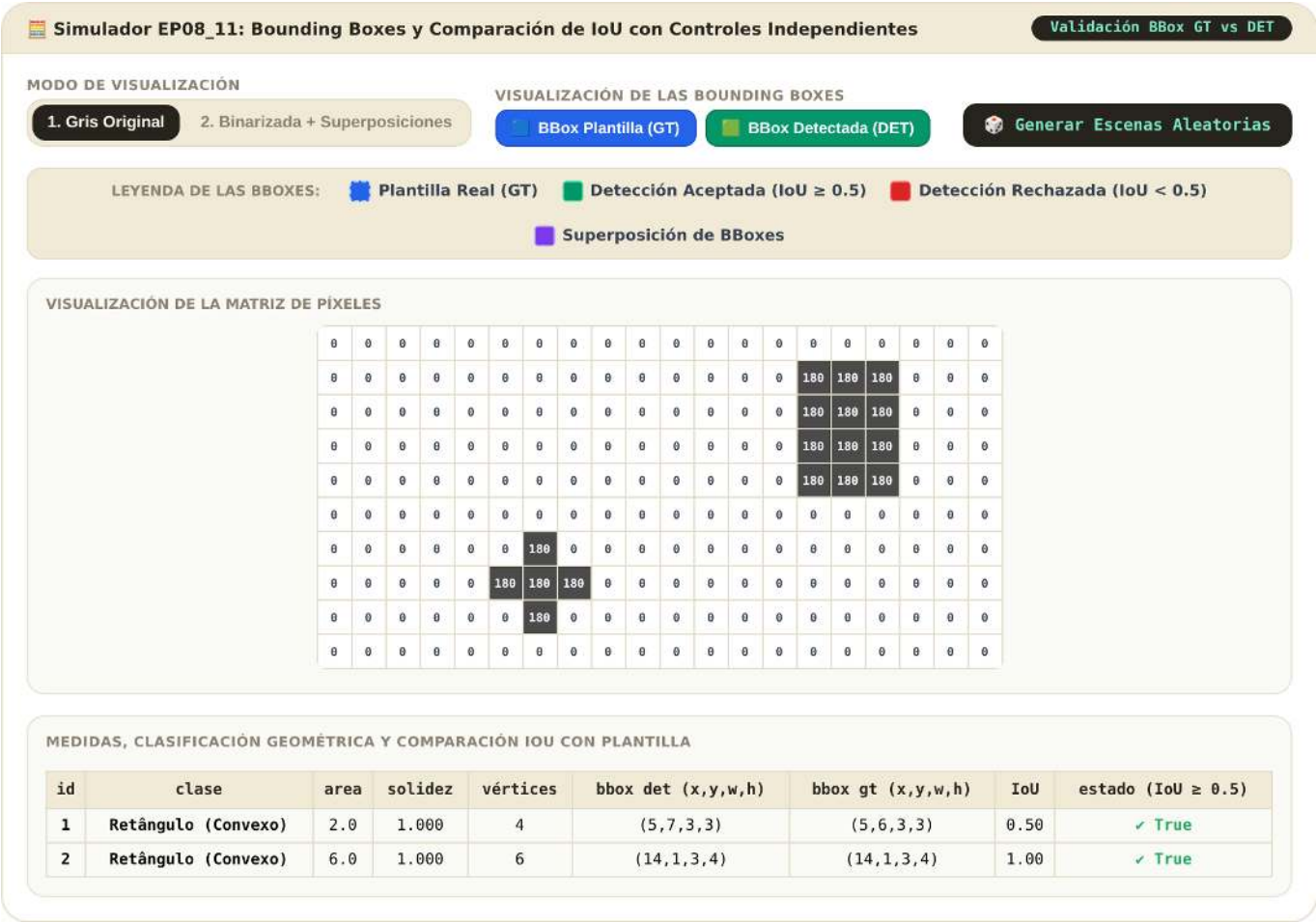
8.14.11.4 📌 Ejemplos

Entrada	Salida
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 0 180 0 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 0 180 180 180 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 2 1 0.10 0.30 0.25 0.60 2 0.60 0.30 0.75 0.60	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 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 1 1 1 0 0 0 0 0 0 0 1 1 1 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 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 hits 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

```
%%writefile EP08_11.py
# Código Python
```

```
Overwriting EP08_11.py
```

```
TestSuite("EP08_11.py").run()
```



id	clase	area	solidez	vértices	bbox det (x,y,w,h)	bbox gt (x,y,w,h)	IoU	estado (IoU ≥ 0.5)
1	Retângulo (Convexo)	2.0	1.000	4	(5,7,3,3)	(5,6,3,3)	0.50	✔ True
2	Retângulo (Convexo)	6.0	1.000	6	(14,1,3,4)	(14,1,3,4)	1.00	✔ True

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap08/sim-ep0811.html

Figura 8.25: Simulador EP08_11: Clasificación Geométrica con Controles Independientes de *Overlays BBox* (GT y DET)

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

8.14.12 EP08_12 🔴 Segmentación de Instancias en Imagen Real: Objetos Geométricos

El ejemplo de segmentación clásica de este capítulo separó "instancias" de monedas por desconexión espacial en la máscara binaria resultante de la umbralización de Otsu. En este ejercicio vas a aplicar la misma idea — pero ahora sobre una imagen real con objetos geométricos variados — encadenando preprocesamiento, binarización, extracción de contornos (`cv2.findContours`) y validación del resultado contra un patrón de *bounding boxes*.

A diferencia del ejercicio anterior (etiquetado sobre máscara ya preparada), aquí partes de la **imagen original**: la calidad de tu segmentación depende directamente de las elecciones de preprocesamiento (filtrado, umbralización, operaciones morfológicas) realizadas antes de etiquetar los componentes.

8.14.12.1 📋 Directrices de Implementación

1. **Entrada:** utilizar la imagen `00000.jpg`.
2. **Preprocesamiento y segmentación:** aplicar las etapas necesarias (filtrado, binarización y operaciones morfológicas) para separar automáticamente los objetos del fondo, sin recortes manuales.
3. **Etiquetado y medición:** para cada objeto segmentado, determinar:
   - área;
   - centro de masa (centroide);
   - tipo, conforme al conjunto `obj2`.
4. **Anotación visual:** escribir, en el interior de cada objeto, su área y la sigla del tipo (`obj2`).
5. **Validación (IoU):** calcular el *Intersection over Union* (IoU) entre el *bounding box* detectado (`cv2.boundingRect`) y el *bounding box* de patrón del tipo correspondiente. Un objeto se considera correctamente segmentado solo si hay exactamente un *bounding box* del tipo correcto con **IoU 0,5**.
6. **Salida:** imprimir, para cada objeto detectado, su identificador, tipo y si fue validado con éxito (`acertou=1`) o no. La impresión debe seguir el orden de las clases de `obj2` (0=Tria … 8=Cruz); dentro de la misma clase, ordenar los objetos por la coordenada vertical del centroide (`cy`) creciente. Al final, imprimir la precisión general.

8.14.12.2 📌 Restricciones Computacionales

- **Sin recorte manual:** toda la segmentación debe realizarse sobre la imagen completa.
- **Conjunto de clases fijo:**

```
obj  = ['Triangulo','Quadrado','Pendagono','Hexagono','Heptagono','Circulo',
        'Elipse','Estrela','Cruz']
obj2 = ['Tria','Quad','Pent','Hexa','Hept','Circ','Elip','Estr','Cruz']
```

- **Dimensión de la imagen:** 608×608 píxeles — usada para desnormalizar las coordenadas del archivo TXT.
- **Validación por centro de masa:** un objeto solo se considera correctamente segmentado si su centroide está estrictamente dentro del *boundbox* del patrón correspondiente al mismo tipo de objeto.

8.14.12.3 🧠 Fundamentación Teórica

Elemento	Papel en la segmentación de instancias
Preprocesamiento (filtrado, umbralización)	Etapa que produce la máscara binaria a partir de la imagen de intensidad original
`cv2.findContours`	Extrae los contornos de los componentes conectados en la máscara binaria

Elemento	Papel en la segmentación de instancias
Momentos geométricos (`cv2.moments`)	Permiten calcular el centro de masa (centroide) de cada contorno
`approxPolyDP` / vértices	Ayuda en la clasificación del tipo de objeto (nº de lados aproximado)
Validación mediante *boundbox*	Confirma si la instancia segmentada corresponde espacialmente a un objeto del patrón, midiendo la precisión del método

8.14.12.4 📌 Ejemplo de Salida Esperada

```
Objeto 1: tipo=Tria, validado=True
...
Precisión: 88.89%
```

Parámetros fijos para reproducibilidad: para que la salida coincida con el patrón de corrección automática, utiliza exactamente: filtro de área mínima de 300 píxeles; `cv2.approxPolyDP` con `epsilon = 0.02 * perímetro`; umbral de solidez 0.92 y conteo de vértices 9 (con 11 para diferenciar Cruz de Estrella) para formas cóncavas; relación de aspecto 1.15 para diferenciar Círculo de Elipse; umbral de IoU = 0.5 en la validación.

8.14.12.5 📌 Archivos de Referencia (`.jpg` y `.txt`)

Para depuración local, se proporcionan dos archivos de referencia (incluidos en esta entrega; al integrarlos al repositorio del capítulo, guárdalos en `all/cap08/dados/EP08/`):

- 📥 **Imagen (`00000.jpg`)**: imagen de objetos geométricos utilizada como entrada del ejercicio. El objetivo es segmentar automáticamente cada objeto, determinar su tipo y calcular sus medidas.
- 📥 **Patrón (`00000.txt`)**: archivo que contiene las *bounding boxes* normalizadas de los objetos presentes en la imagen. Cada línea posee el identificador de la clase y las coordenadas normalizadas de las esquinas superior izquierda e inferior derecha, utilizándose para validar automáticamente la segmentación.

La Figura 8.26 presenta la imagen de entrada y la misma imagen con las *bounding boxes* dibujadas a partir del archivo de patrón.

```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"Baixando {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=[
        "Imagem original",
        "Bounding boxes del gabarito"
    ],
    cols=2,
    figsize=(10,5)
)
```

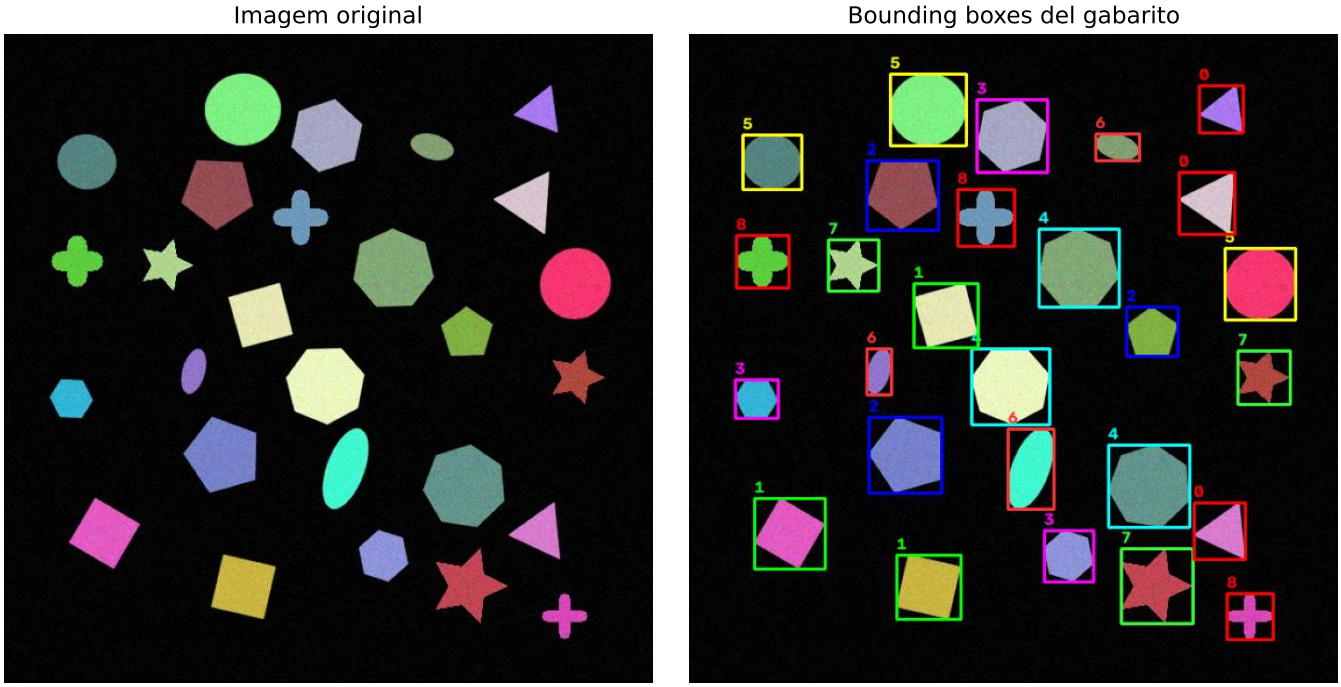


Figura 8.26: Simulador EP08_12: Imagem utilizada no EP08_12. À esquerda, a imagem original. À direita, a imagem com as *bounding boxes* do arquivo de gabarito.

```
%%writefile EP08_12.py
# Código Python
```

```
Overwriting EP08_12.py
```

```
TestSuite("EP08_12.py").run()
```

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

Simulador EP08_12: Precisión de Segmentación en Múltiples Objetos

acertó si IoU ≥ umbral Y tipo correcto

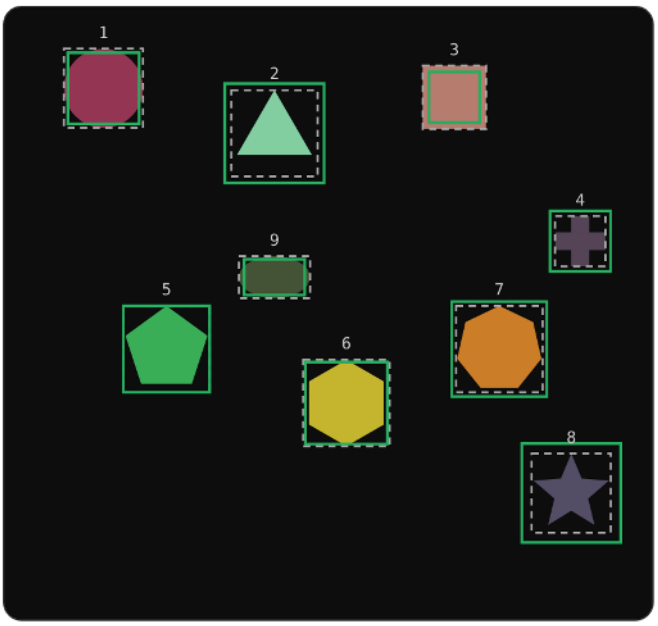
Cada forma tiene un *boundbox* de referencia (rectángulo discontinuo, justo alrededor de la forma) y un *boundbox* detectado (rectángulo sólido, desplazado/ruidoso). Ajuste el ruido, el sesgo y el umbral de IoU para ver cómo cambia la validación.

ruido de segmentación (px, jitter máx. por lado) 0

sesgo sistemático en x (px) 0

umbral de IoU 0.50

simular error de clasificación (2 objetos con tipo intercambiado)



id	tipo real	tipo detectado	IoU	≥ umbral	acertó
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.es/cap08/sim-ep0812.html

Figura 8.27: Simulador EP08_12: Precisión de Segmentación en Múltiples Objetos (IoU)

Capítulo 9

Aprendizaje Profundo para Visión por Computador

Executar Colab Abrir GitHub

Los capítulos anteriores establecieron los fundamentos de la Visión por Computador (VC) mediante métodos clásicos de extracción y representación de características. En el **Capítulo 7**, descriptores como *Local Binary Patterns* (*LBP*) e *Histogram of Oriented Gradients* (*HOG*) mostraron cómo texturas y formas pueden codificarse mediante descriptores diseñados manualmente. En el **Capítulo 8**, algoritmos como *Oriented FAST and Rotated BRIEF* (*ORB*) y el detector *Haar Cascade* extendieron ese principio a tareas de correspondencia, detección y reconocimiento de objetos.

Estas técnicas siguen siendo relevantes por su interpretabilidad y eficiencia computacional, pero dependen de una etapa previa de definición manual de descriptores, denominada **ingeniería de características** (*feature engineering*). Esa dependencia limita la adaptación del modelo a escenarios para los cuales el descriptor no fue diseñado.

El **Aprendizaje Profundo** (*Deep Learning*) propone una alternativa: en lugar de especificar manualmente las características relevantes, el modelo aprende automáticamente representaciones a partir de los datos durante el entrenamiento —proceso conocido como **aprendizaje de representaciones** (*representation learning*) (GOODFELLOW, 2016). En la VC, esta estrategia se implementa principalmente mediante las **Redes Neuronales Convolucionales** (*Convolutional Neural Networks* — CNNs), en las cuales los filtros convolucionales dejan de tener coeficientes fijos y pasan a ser ajustados por algoritmos de optimización.

Aunque representan un cambio en la construcción de sistemas de reconocimiento de patrones, las CNNs preservan conceptos ya estudiados en este libro: la convolución, presentada en el **Capítulo 3**, sigue siendo la operación responsable de la extracción local de características, ahora aplicada con coeficientes aprendidos en lugar de diseñados.

Cabe señalar que el objetivo de este capítulo no es explorar exhaustivamente la teoría del Aprendizaje Profundo, sino ofrecer una visión general de sus fundamentos y demostrar cómo se aplican estas arquitecturas en el contexto de la VC. Los lectores interesados en una profundización teórica y conceptual en el área deben recurrir a referencias especializadas de la literatura, como Goodfellow (2016) y Lecun (2015).

9.1 Objetivos del Capítulo

Al final de este capítulo, el estudiante deberá ser capaz de:

- Relacionar la convolución aprendida por las CNNs con la convolución de *kernels* fijos presentada en el **Capítulo 3**;
- Describir la arquitectura básica de una CNN y la función de sus capas principales;
- Implementar, entrenar y evaluar modelos de CNN para clasificación de imágenes;
- Aplicar **transferencia de aprendizaje** (*transfer learning*) para adaptar modelos preentrenados a nuevos problemas;
- Utilizar modelos preentrenados en tareas de clasificación, detección de objetos y segmentación;

- Implementar, entrenar y evaluar una arquitectura *U-Net* para segmentación semántica, comparándola con enfoques clásicos;
- Preparar conjuntos de datos anotados e integrarlos en un *pipeline* de entrenamiento mediante plataformas como **Roboflow**;
- Integrar geometría computacional y Aprendizaje Profundo en aplicaciones de realidad aumentada y fotogrametría.

La Figura 9.1 sintetiza la organización de los conceptos estudiados en este capítulo y las relaciones entre ellos.

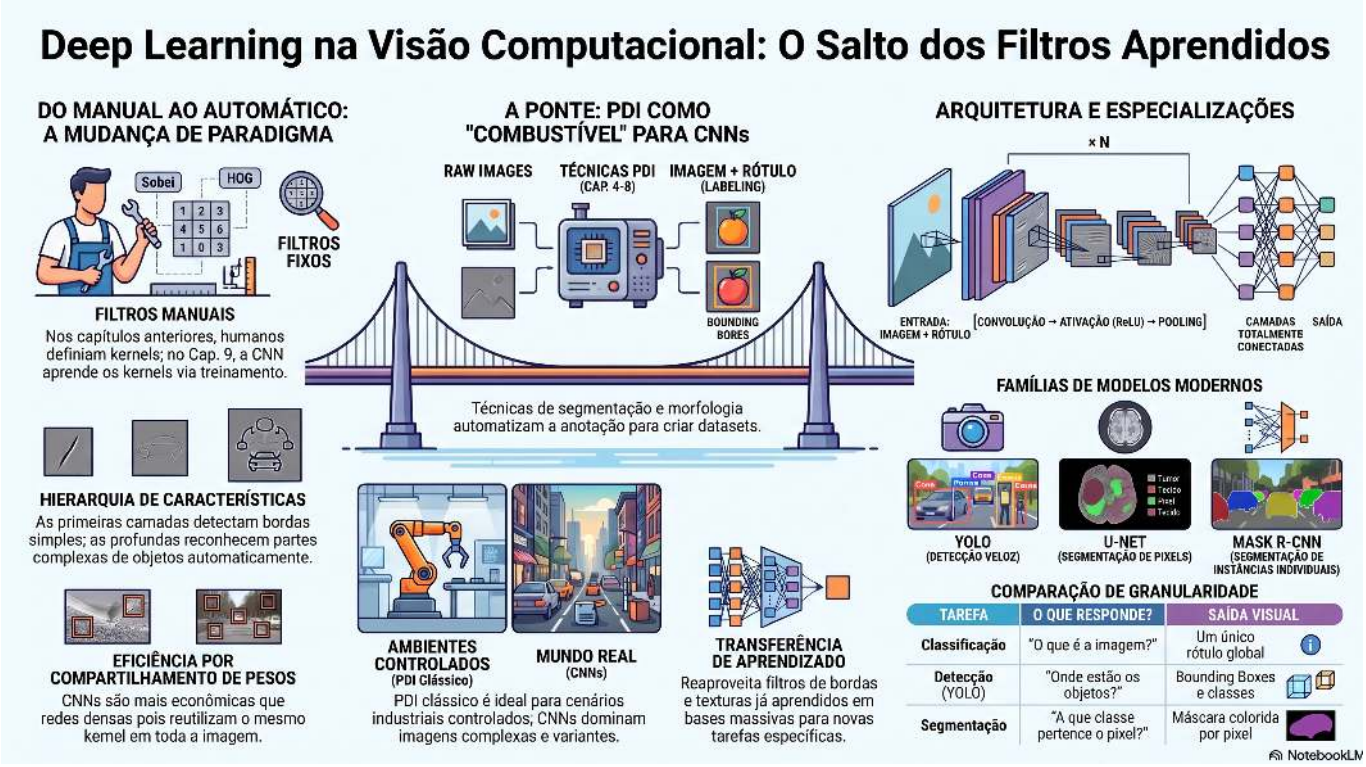


Figura 9.1: Visión general de los principales conceptos abordados en este capítulo. **Fuente:** elaborado con ayuda de *Gemini Notebook* ({GOOGLE}, 2025).

9.2 Panorama: Clasificación, Detección y Segmentación

Las tareas de VC se diferencian, sobre todo, por la información producida como salida. La **clasificación** asigna una única etiqueta a la imagen completa; la **detección de objetos** localiza e identifica los objetos presentes en la escena; la **segmentación** asocia una clase a cada píxel y, en algunos enfoques, distingue diferentes instancias de una misma categoría.

La Tabla 9.1 resume las tareas estudiadas a lo largo del libro, indicando la pregunta que cada una responde y la granularidad de la información producida.

Tabla 9.1: Comparativa entre las principales tareas de VC según la granularidad de la información producida.

Tarea	Pregunta respondida	Granularidad de la salida	Capítulo
Clasificación	"¿Cuál es la clase de esta imagen?"	Una única etiqueta para la imagen completa	7 y 9
Detección de objetos	"¿Qué objetos existen y dónde están?"	Clase y caja delimitadora (*bounding box*) para cada objeto	8 y 9
Segmentación semántica	"¿A qué clase pertenece cada píxel?"	Una etiqueta de clase para cada píxel	8 y 9

Tarea	Pregunta respondida	Granularidad de la salida	Capítulo
Segmentación de instancias	"¿Qué píxeles pertenecen a cada objeto?"	Una etiqueta por píxel para cada instancia	8 y 9
Segmentación panóptica	"¿Cuál es la clase y la identidad de cada objeto?"	Clase e identificador de instancia para cada píxel	8 y 9

Estas tareas representan niveles crecientes de interpretación de la imagen: la clasificación describe la escena de forma global, la detección añade la localización de los objetos y la segmentación produce una representación espacial detallada, permitiendo analizar cada región individualmente. Este capítulo se centra, primero, en la clasificación por CNNs, para luego extender los mismos principios a la detección y a la segmentación.

9.3 Configuración del Entorno

Los ejemplos de este capítulo utilizan **PyTorch**, un *framework* ampliamente empleado en el desarrollo y entrenamiento de modelos de Aprendizaje Profundo. El siguiente código verifica la disponibilidad de las bibliotecas necesarias e instala, automáticamente, aquellas aún no presentes en el entorno de ejecución.

En caso de que **PyTorch** no esté instalado, se selecciona automáticamente una versión compatible con el hardware disponible: la versión con soporte para **CUDA**, si hay una GPU NVIDIA disponible, o la versión para ejecución en CPU, en caso contrario.

A continuación, el entorno se inicializa con la importación de las bibliotecas utilizadas a lo largo del capítulo, la definición de una semilla aleatoria para favorecer la reproducibilidad de los experimentos y la obtención del archivo `morph.py` — la biblioteca didáctica de procesamiento morfológico ya utilizada en capítulos anteriores —, en caso de que aún no esté disponible en el directorio de trabajo.

```
import contextlib, importlib, importlib.metadata, importlib.util
import io, os, random, shutil, subprocess, sys, urllib.request, warnings

# Suprime avisos de PyTorch y advertencias generales
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

# Silencia stdout y stderr tanto a nivel de Python como de descriptores de archivo del SO
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
```

```python
def setup_cap09():
    """Instala librerías faltantes para este capítulo de forma 100% silenciosa."""
    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"  Ambiente listo. OpenCV {cv2.__version__} | "
      f"morph {getattr(mm, '__version__', 'local_file')} | "
      f"PyTorch {torch.__version__} | Ultralytics {ver} | {device}{gpu}")
```

```
  Ambiente listo. OpenCV 5.0.0 | morph local_file | PyTorch 2.6.0+cu124 | Ultralytics 8.4.113
| cuda (NVIDIA GeForce GTX TITAN X)
```

9.4 Fundamentos del Aprendizaje Profundo para VC

Las CNN son la principal arquitectura de Aprendizaje Profundo aplicada al análisis de imágenes. Su funcionamiento se basa en la composición de operaciones convolucionales organizadas en capas sucesivas, en las cuales los filtros aprendidos durante el entrenamiento transforman la imagen en representaciones progresivamente más abstractas. En esta sección, se presentan los conceptos fundamentales que conectan la convolución espacial estudiada anteriormente con los modelos modernos de VC, incluyendo la extracción jerárquica de características, el proceso de entrenamiento y la utilización de modelos preentrenados.

9.4.1 De la Convolución Fija a la Convolución Aprendida

El **Capítulo 3** presentó la convolución espacial con *kernels* fijos, como los operadores de Sobel, diseñados para resaltar características específicas de una imagen. En los **Capítulos 7** y **8**, el mismo principio sustentó descriptores como *HOG*, *LBP* y *ORB*, además del detector *Haar Cascade*: en todos estos casos, los filtros se definen antes de la ejecución del algoritmo y permanecen inalterados durante el procesamiento.

La Figura 9.2 retoma el funcionamiento de la convolución espacial: el simulador permite seleccionar diferentes *kernels* y seguir el desplazamiento de la ventana de convolución sobre una imagen. En cada posición, los coeficientes del *kernel* se combinan con la vecindad local de la imagen —denominada **campo receptivo** (*receptive field*)— para producir un valor del **mapa de características** (*feature map*), ilustrando también el compartimiento de pesos (*weight sharing*).

Las CNNs preservan esta operación, pero sustituyen *kernels* fijos por **filtros aprendidos**: en lugar de coeficientes definidos previamente, la red ajusta estos valores durante el entrenamiento a partir de ejemplos etiquetados, buscando minimizar una **función de pérdida** (*loss function*), que mide la diferencia entre las predicciones del modelo y las respuestas esperadas.

La diferencia esencial entre los métodos clásicos y las CNNs, por tanto, no reside en la operación de convolución en sí, sino en la forma en que se obtienen los filtros: mientras que los primeros utilizan filtros diseñados manualmente, las CNNs aprenden, a partir de los datos de entrenamiento, representaciones adecuadas para la tarea.

Para comprender cómo ocurre ese aprendizaje, es necesario estudiar la unidad básica de procesamiento de las redes neuronales: el **neurona artificial**.

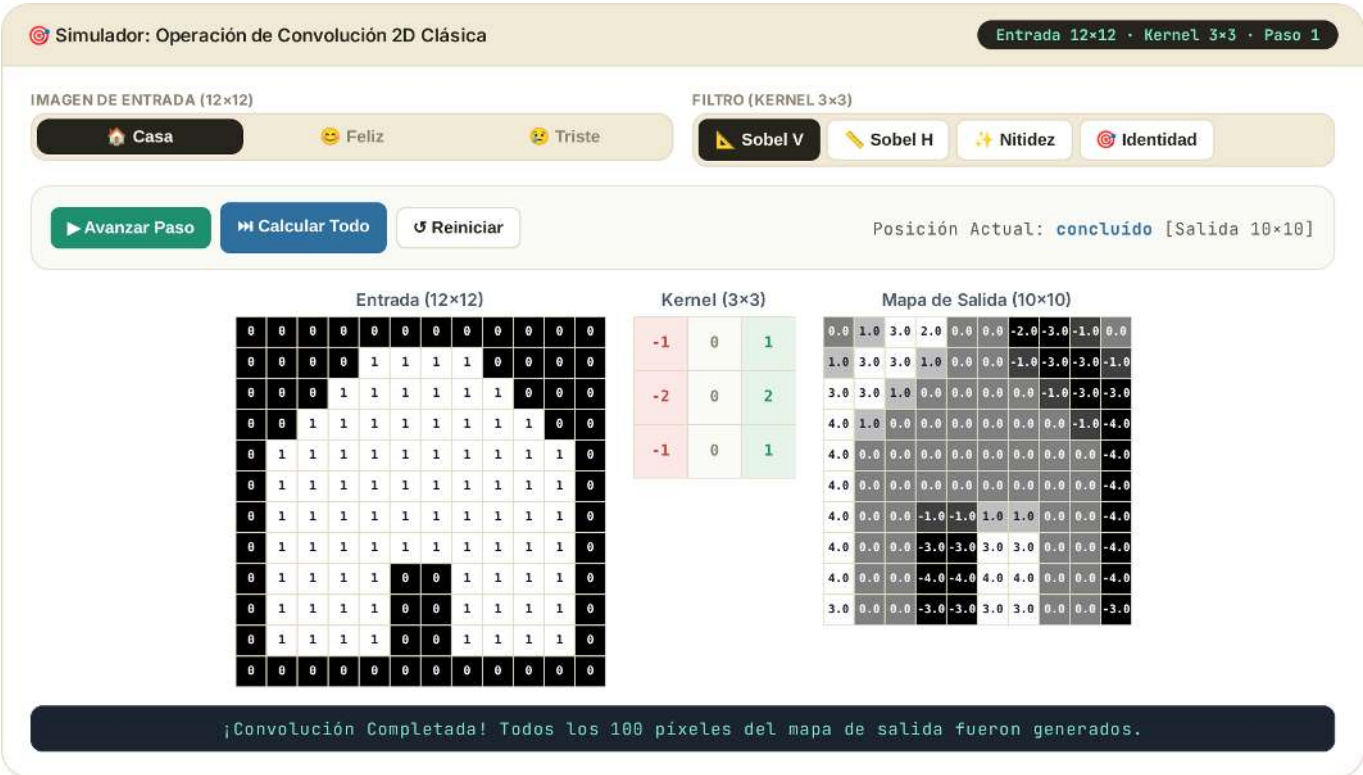
9.4.2 Neurona Artificial

La **neurona artificial** (*artificial neuron*) es la unidad fundamental de procesamiento de una red neuronal. Su primer modelo matemático —un conjunto de entradas combinadas y comparadas con un umbral— fue propuesto por Mcculloch (1943), aún sin ningún mecanismo de aprendizaje. El **Perceptrón** (ROSENBLATT, 1958) avanzó sobre esta formulación al introducir una regla de ajuste de los pesos a partir de ejemplos, convirtiéndose en el primer modelo de neurona artificial capaz de aprender y en la base de las arquitecturas modernas de **Aprendizaje Profundo** (*Deep Learning*). El término Aprendizaje Profundo se refiere al uso de redes con múltiples capas de procesamiento, capaces de aprender representaciones jerárquicas de los datos: las primeras capas aprenden características simples, como bordes y texturas, y las capas más profundas combinan progresivamente estas representaciones para identificar estructuras y objetos más complejos.

Cada neurona recibe un conjunto de entradas, calcula una combinación lineal de estos valores y aplica una **función de activación** (*activation function*), produciendo un único valor de salida. Matemáticamente, la combinación lineal está dada por

$$z = \sum_{i=1}^{n} w_i x_i + b,$$

donde x_i representan las entradas, w_i los pesos asociados a cada entrada y b el **sesgo** (*bias*). La salida de la neurona se obtiene aplicando la función de activación:



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-09-convolucao.html

Figura 9.2: Simulador interactivo de convolución 2D clásica: elige entre las tres imágenes sintéticas de entrada 12×12 (casa, cara feliz o triste) y un filtro 3×3 (Sobel V, Sobel H, Nitidez o Identidad) y avanza paso a paso para observar cómo los productos internos locales del campo receptivo construyen el mapa de características célula a célula.

$$y = f(z).$$

En las CNNs, este principio adopta formas distintas según la capa. En las **capas convolucionales** (*convolutional layers*), cada neurona procesa solo una pequeña región de la entrada, denominada **campo receptivo** (*receptive field*), preservando la organización espacial de la imagen. En las **capas totalmente conectadas** (*fully connected layers*), cada neurona recibe todas las salidas de la capa anterior, combinando las características extraídas para producir la salida final de la red, como la clase asignada a la imagen.

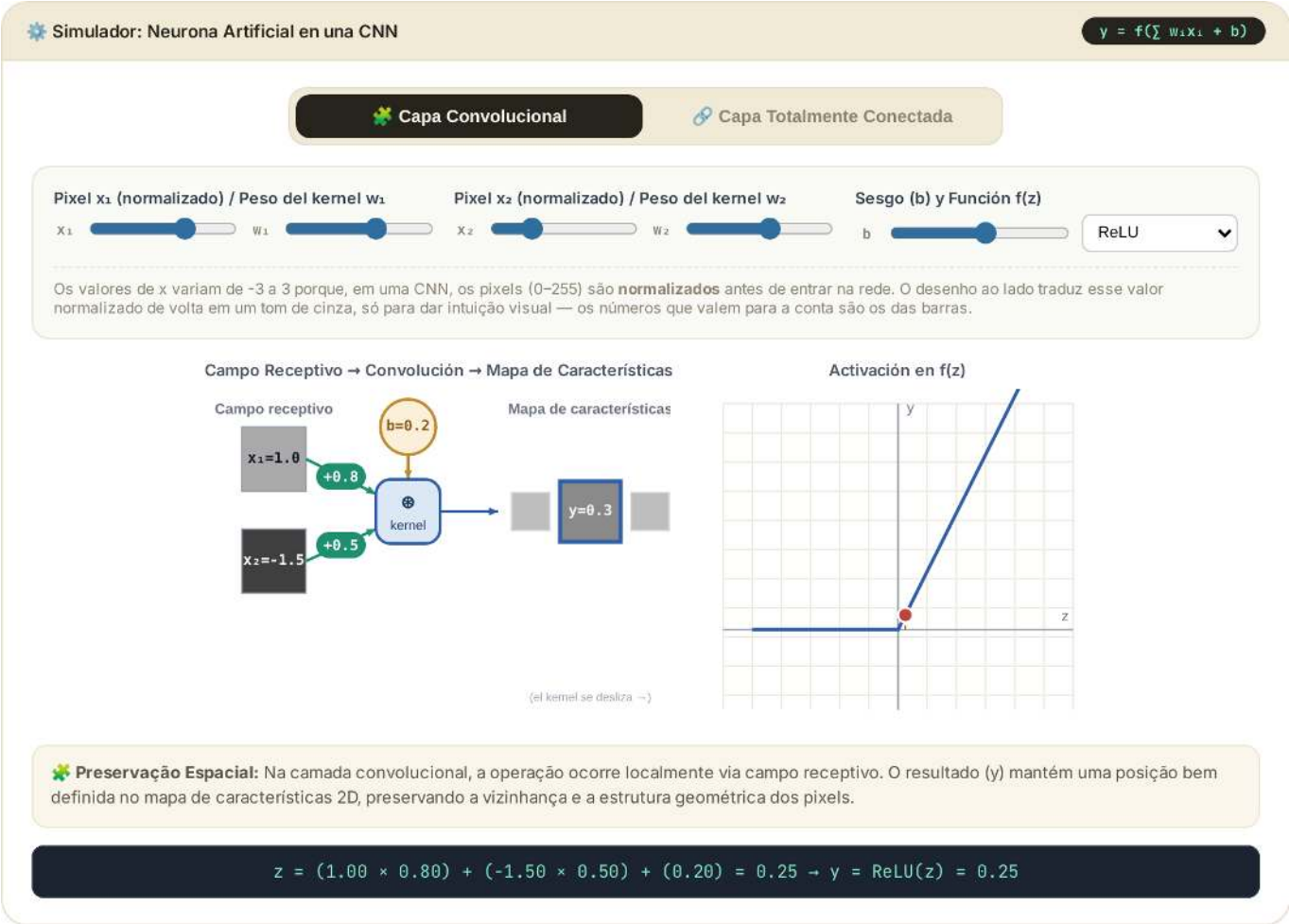
La Figura 9.3 ilustra el funcionamiento de una neurona artificial: el simulador permite modificar las entradas (x_1 y x_2), los pesos (w_1 y w_2), el sesgo (b) y la función de activación, observando en tiempo real el cálculo de la combinación lineal y de la salida correspondiente.

En una CNN, miles de neuronas se organizan en capas con funciones específicas: las primeras son responsables de la extracción de características mediante la convolución, y las últimas realizan la clasificación a partir de las características aprendidas.

9.4.3 Capa Convolucional

La **capa convolucional** (*convolutional layer*) es responsable de la extracción de características de la imagen. Cada filtro genera un **mapa de características** (*feature map*), cuya intensidad en cada posición indica la respuesta del filtro a la región correspondiente de la entrada.

La operación realizada sigue el mismo principio de desplazamiento y combinación local presentado en el **Capítulo 3** para la convolución espacial. Considerando un *kernel* K de dimensión $k \times k$, el valor producido en la posición (i, j) está dado por



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-09-neuronio.html

Figura 9.3: Simulador interactivo de la Neurona Artificial en contexto de CNN: alterne entre una neurona de capa convolucional (donde x_i son intensidades de píxel en un campo receptivo y w_i son pesos del *kernel*) y una neurona de capa totalmente conectada, ajustando entradas, pesos, sesgo y función de activación para visualizar el cálculo de z y de la salida y en tiempo real.

$$F(i,j) = \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} K(u,v)\, I(i+u, j+v).$$

Cabe registrar una distinción terminológica: la expresión anterior corresponde, formalmente, a una **correlación cruzada** (*cross-correlation*), y no a la convolución matemática estricta, que exige la reflexión del *kernel* antes de la combinación. La mayoría de los *frameworks* de Aprendizaje Profundo, incluido **PyTorch**, implementa esta operación sin reflexión y la designa, por convención, como convolución — convención adoptada también en este capítulo. Esta diferencia no tiene efecto práctico sobre el entrenamiento, ya que los coeficientes del *kernel* se aprenden y no se imponen previamente.

La principal diferencia respecto a los métodos clásicos radica, así, en la obtención del *kernel* K: en filtros tradicionales, sus coeficientes se definen manualmente para resaltar características específicas de la imagen; en las CNNs, los coeficientes se inicializan automáticamente y se ajustan durante el entrenamiento mediante la **retropropagación del error** (*backpropagation*), lo que hace que cada filtro se especialice en identificar patrones relevantes para la tarea en estudio.

Dos conceptos caracterizan esta capa:

- **Compartición de pesos** (*weight sharing*): el mismo filtro se aplica en todas las posiciones de la imagen, reduciendo significativamente el número de parámetros del modelo.
- **Campo receptivo** (*receptive field*): cada neurona convolucional procesa solo una pequeña vecindad de la imagen, preservando la estructura espacial de los datos.

Al apilar varias capas convolucionales, la red aprende una **jerarquía de características**: las primeras capas tienden a detectar patrones simples, como bordes y texturas, y las capas más profundas combinan esta información para representar estructuras progresivamente más complejas. Tras la convolución, el mapa de características se somete a una función de activación, introduciendo no linealidad en el modelo y ampliando su capacidad para representar relaciones complejas entre las variables de entrada.

La Figura 9.4 presenta esta capa de forma interactiva.

9.4.4 Función de Activación

La convolución es una operación lineal. Para que la red pueda modelar relaciones no lineales entre entradas y salidas, se aplica una **función de activación** (*activation function*) tras cada capa convolucional.

La función más utilizada en CNNs es la **ReLU** (*Rectified Linear Unit*), definida por

$$\mathrm{ReLU}(x) = \max(0, x).$$

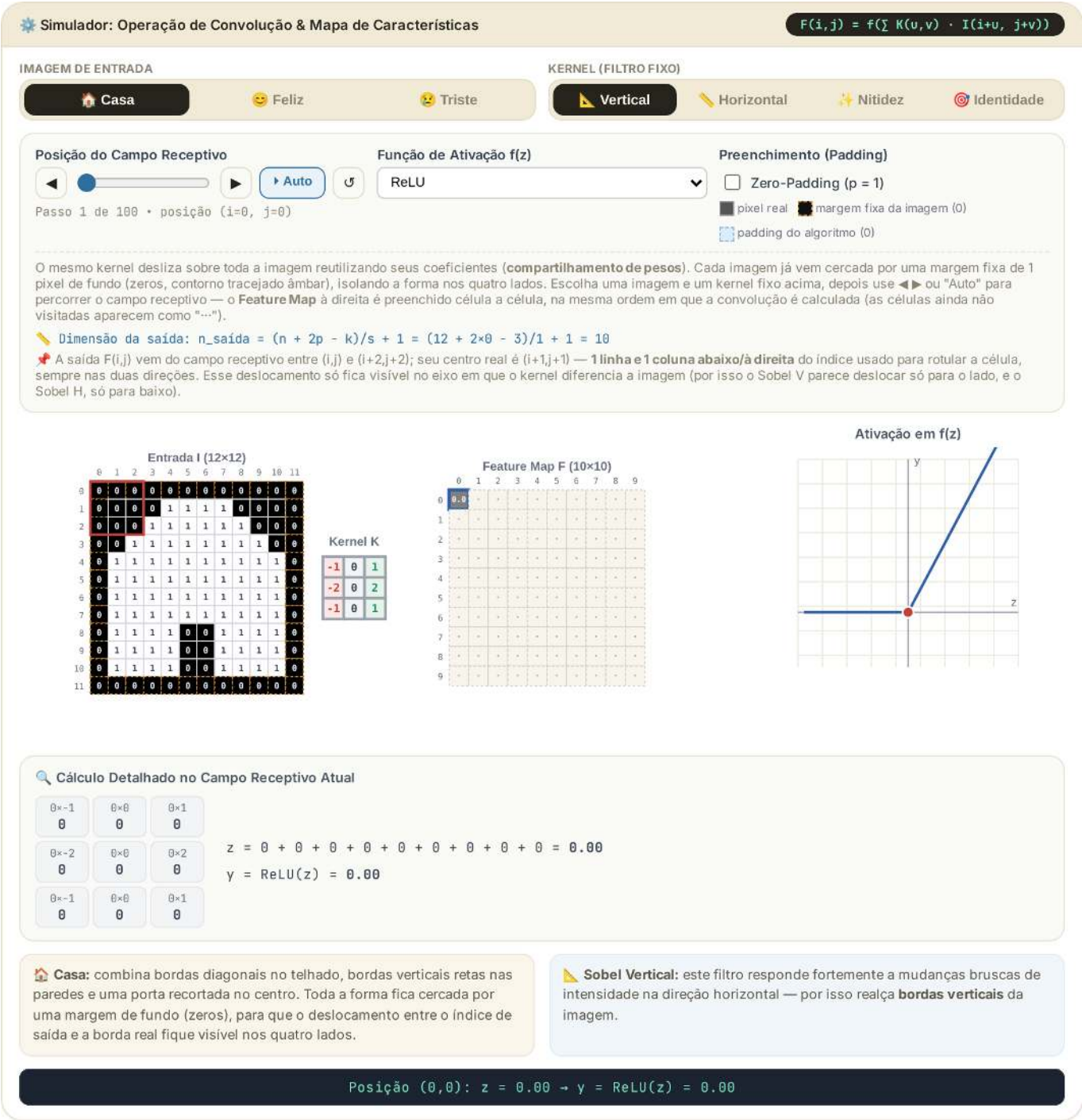
Esta función preserva los valores positivos y reemplaza por cero los valores negativos, introduciendo no linealidad en el modelo y favoreciendo el entrenamiento de redes profundas con bajo costo computacional.

La Figura 9.5 ilustra el funcionamiento de la **ReLU** aplicada tanto a valores individuales como a un mapa de características, permitiendo comparar la salida antes y después de la activación.

Los mapas de características resultantes de la convolución y de la activación preservan la estructura espacial de la imagen. En muchas arquitecturas, la etapa siguiente reduce su resolución mediante una operación de *pooling*.

9.4.5 *Pooling*

La capa de ***pooling*** reduce la resolución espacial de los mapas de características, preservando la información más relevante para las etapas siguientes del procesamiento. La operación más utilizada es el ***max-pooling***, que selecciona el mayor valor en cada ventana de la imagen:



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-09-camada-conv.html

Figura 9.4: Simulador interativo da Camada Convolucional: escolha entre três imagens de entrada 12×12 (casa, rosto feliz ou rosto triste) para observar como os mesmos *kernels* fixos reagem a diferentes bordas e formas. Navegue pelo campo receptivo com os botões ou reprodução automática, ajuste a função de ativação e alterne o zero-padding, acompanhando o mapa de características sendo revelado célula a célula, com o cálculo detalhado termo a termo e a fórmula da dimensão de saída em tempo real.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-09-relu.html

Figura 9.5: Simulador interactivo de la función de activación ReLU: arrastre el control para ver cómo los valores negativos se ponen a cero y los valores positivos se conservan, tanto en la curva como en un mapa de características real.

$$P(i,j) = \max_{(u,v)\in \text{janela}(i,j)} F(u,v).$$

Esta reducción disminuye el costo computacional de las capas subsiguientes y hace que la representación sea más robusta ante pequeñas variaciones en la posición de los patrones presentes en la imagen.

La Figura 9.6 presenta esta operación sobre un mapa de características de 8×8 píxeles, reducido a 4×4 mediante ventanas de 2×2 con paso igual a 2, alternando entre **max-pooling** y **average-pooling** — que calcula, en lugar del máximo, el promedio de los valores de la ventana correspondiente.

En conjunto, convolución, función de activación y *pooling* forman el bloque básico utilizado en la construcción de una CNN.

9.4.6 Entrenamiento de Redes Neuronales: Cómo Aprenden las CNNs

Una CNN aprende ajustando automáticamente sus parámetros — los coeficientes de los filtros convolucionales, los pesos de las capas totalmente conectadas y los sesgos (*biases*) — a partir de ejemplos etiquetados. Este entrenamiento es iterativo e involucra tres etapas: medir el error producido por la red mediante una **función de pérdida** (*loss function*), calcular cómo depende ese error de cada parámetro mediante la **retropropagación** (*backpropagation*) y actualizar los parámetros con un **algoritmo de optimización** (*optimizer*).

9.4.6.1 Función de Pérdida (*Loss Function*)

La **función de pérdida** (*loss function*) cuantifica la diferencia entre la predicción de la red y la respuesta correcta, denominada **verdad de referencia** (*ground truth*). El resultado es un escalar L: cuanto menor es la pérdida, más cercana está la predicción a la respuesta esperada.

En problemas de clasificación multiclase, la función más utilizada es la **Entropía Cruzada** (*Cross-Entropy Loss*), aplicada a las probabilidades producidas por la capa **Softmax**:

$$L = -\sum_{c=1}^{C} y_c \log(\hat{y}_c),$$

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-09-pooling.html

Figura 9.6: Simulador interactivo de *pooling*: elija entre *max-pooling* y *average-pooling* y avance paso a paso para observar la reducción de la resolución espacial del mapa de características.

donde C es el número de clases, y_c es la etiqueta real en codificación *one-hot* y $\hat{y}_c$ es la probabilidad prevista para la clase c. La pérdida se aproxima a cero cuando la red asigna alta probabilidad a la clase correcta y crece rápidamente a medida que esa probabilidad disminuye.

La Figura 9.7 ilustra este comportamiento: el simulador permite seleccionar la clase correcta y alterar las probabilidades producidas por la *Softmax*, mostrando en tiempo real la variación de la función de pérdida.

9.4.6.2 Retropropagación (*Backpropagation*)

Tras el cálculo de la pérdida, es necesario determinar cómo contribuye cada parámetro de la red a dicho resultado. Esta etapa se lleva a cabo mediante la **retropropagación** (*backpropagation*), que aplica la **Regla de la Cadena** del cálculo diferencial para obtener el gradiente de la función de pérdida con respecto a cada parámetro.

Para un parámetro w, este gradiente se expresa como

$$\frac{\partial L}{\partial w}.$$

El gradiente indica cómo varía la pérdida ante pequeños cambios en w: un gradiente positivo señala que aumentar w incrementa la pérdida, mientras que un gradiente negativo refleja el efecto opuesto.

La Figura 9.8 presenta este proceso de forma visual, mostrando la propagación del gradiente desde la capa de salida hasta las primeras capas convolucionales.

9.4.6.3 Algoritmos de Optimización

Tras el cálculo de los gradientes, un **algoritmo de optimización** (*optimizer*) actualiza los parámetros de la red para reducir la función de pérdida. En redes profundas, esta búsqueda ocurre en un espacio de alta dimensión y, en general, **no convexo**, lo que convierte la optimización en un problema desafiante.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-09-loss.html

Figura 9.7: Simulador interactivo de la Función de Pérdida (*Cross-Entropy*): seleccione la clase real de la imagen (Casa, Feliz o Triste) y ajuste las probabilidades estimadas por la Softmax para visualizar el cálculo de la penalización escalar y el gráfico del logaritmo negativo en tiempo real.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-09-backprop.html

Figura 9.8: Simulador interactivo de *Backpropagation*: avance los pasos de la Regla de la Cadena para acompañar el flujo de la señal de error en el sentido inverso de la red, observando el cálculo de las derivadas parciales del gradiente en cada capa.

Para facilitar la comprensión, la Figura 9.9 utiliza una **superficie de pérdida simplificada**, con un mínimo global, un mínimo local y una barrera entre estas regiones. El **mínimo global** corresponde al menor valor de la función de pérdida y representa el mejor conjunto de parámetros de la red; un **mínimo local** también presenta baja pérdida, pero puede estar lejos de la mejor solución. Cuando la optimización queda atrapada en un mínimo local, los ajustes de filtros, pesos y sesgos se vuelven muy pequeños, y el entrenamiento se detiene antes de alcanzar un modelo con menor error.

Descenso de Gradiente Estocástico (*SGD*)

El **Descenso de Gradiente Estocástico** (*Stochastic Gradient Descent — SGD*) actualiza los parámetros en la dirección opuesta al gradiente:

$$w_{\text{novo}} = w_{\text{atual}} - \eta \frac{\partial L}{\partial w},$$

donde η es la **tasa de aprendizaje** (*learning rate*), responsable de controlar el tamaño de la actualización. El *SGD* utiliza únicamente el gradiente de la iteración actual; cuando la búsqueda alcanza un mínimo local, los gradientes se vuelven muy pequeños y las actualizaciones prácticamente cesan.

Optimizadores Adaptativos: *Adam*

El **Adam** (*Adaptive Moment Estimation*) combina estimaciones adaptativas de los primeros y segundos momentos de los gradientes (KINGMA, 2015), adaptando la tasa de aprendizaje de cada parámetro individualmente. Esta adaptación favorece, en muchos casos, la superación de mínimos locales que retendrían al *SGD*.

La Figura 9.9 compara la trayectoria del *SGD* y del *Adam* sobre la misma superficie de pérdida no convexa.

9.4.7 Arquitectura de una CNN

Una CNN para clasificación de imágenes combina las capas presentadas en las secciones anteriores. Durante el **paso hacia adelante** (*forward pass*), la imagen recorre sucesivamente las capas convolucionales, las funciones de activación, las operaciones de *pooling*, la etapa de **Flatten**, las capas totalmente conectadas y, por último, la capa **Softmax**, que produce las probabilidades de las clases. Durante el entrenamiento, esta predicción se compara con la etiqueta correcta para calcular la función de pérdida, realizar la retropropagación y actualizar los parámetros mediante un algoritmo de optimización.

La Figura 9.10 presenta este flujo de procesamiento y entrenamiento.

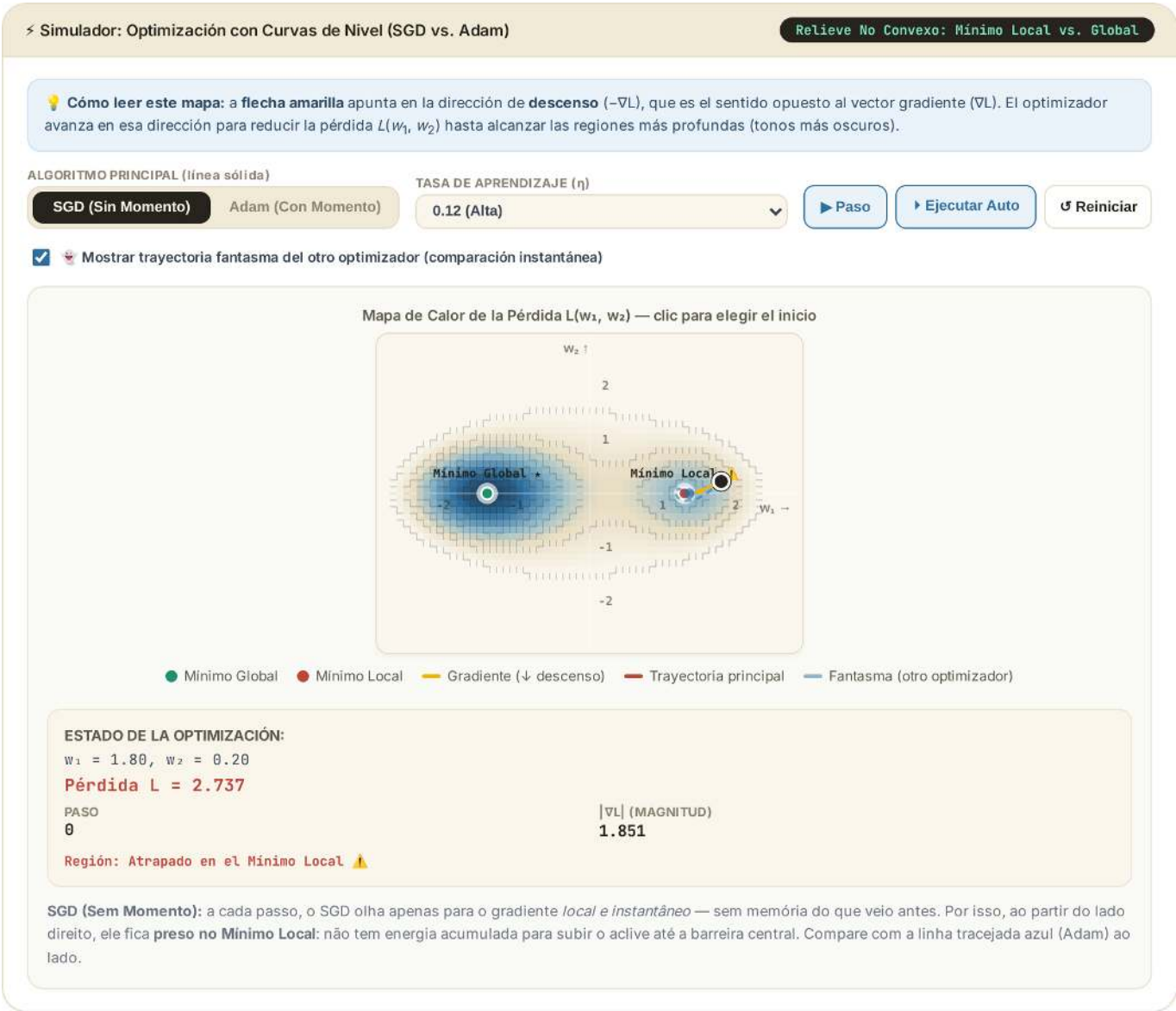
Después del último bloque convolucional, la operación **Flatten** reorganiza los mapas de características en un vector unidimensional, que alimenta las **capas totalmente conectadas** (*fully connected layers*), responsables de combinar las características extraídas para producir los puntajes (*logits*) de cada clase. La capa **Softmax** convierte estos puntajes en una distribución de probabilidades, utilizada tanto para la clasificación como para el cálculo de la función de pérdida durante el entrenamiento.

La Figura 9.11 presenta una versión interactiva de esta arquitectura, permitiendo ejecutar sucesivas etapas de entrenamiento y observar la reducción de la pérdida, la retropropagación de los gradientes y la actualización de los filtros de la red.

9.4.8 Cómo el Gradiente Ajusta los *Kernels* de la Convolución

La comprensión del proceso de aprendizaje en una Red Neuronal Convolucional (CNN) requiere la elucidación de un mecanismo fundamental: **cómo los coeficientes aleatorios de un filtro inicial se transforman en detectores precisos de bordes, texturas y patrones complejos.**

La respuesta reside en el principio del **compartimiento de pesos (*weight sharing*)**. Durante la etapa de propagación hacia adelante (*forward pass*), el mismo filtro de dimensión 3×3 se desliza por toda la extensión



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-09-opt.html

Figura 9.9: Simulador interactivo de los Algoritmos de Optimización: compare la trayectoria del SGD y del Adam sobre una superficie de pérdida no convexa con mapa de calor y curvas de nivel. La línea sólida muestra el optimizador seleccionado avanzando paso a paso; la línea discontinua muestra, para comparación instantánea, el camino completo que el otro optimizador recorrería desde el mismo punto inicial. Observe cómo el SGD queda atrapado en el Mínimo Local a la derecha, mientras que el Adam puede o no cruzar la barrera central dependiendo del impulso acumulado y de la tasa de aprendizaje. Haga clic en cualquier punto del mapa para restablecer el punto inicial de los pesos.

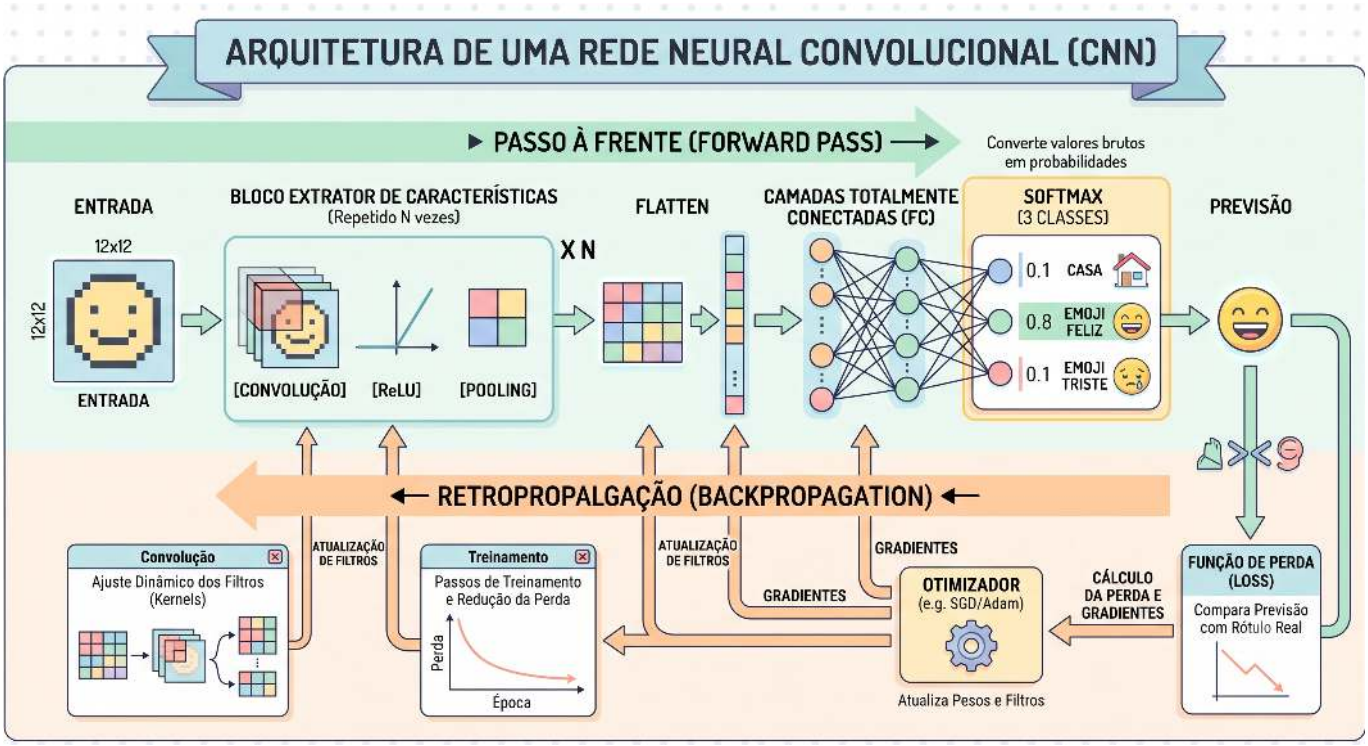


Figura 9.10: Arquitectura simplificada de una CNN para clasificación de imágenes, destacando el *forward pass* y las etapas de entrenamiento mediante la función de pérdida, la retropropagación y el algoritmo de optimización.

de la imagen de entrada. En consecuencia, cada peso del *kernel* — como el elemento $K[0][0]$ en la esquina superior izquierda — se reutiliza múltiples veces a lo largo de las diferentes regiones espaciales del dato de entrada.

Durante la etapa de retropropagación (*backpropagation*), esa reutilización establece una dinámica directa: **cada posición espacial procesada por el filtro genera una contribución individual ("voto") para la actualización del respectivo peso.**

9.4.8.1 La Intuición Detrás del Cálculo

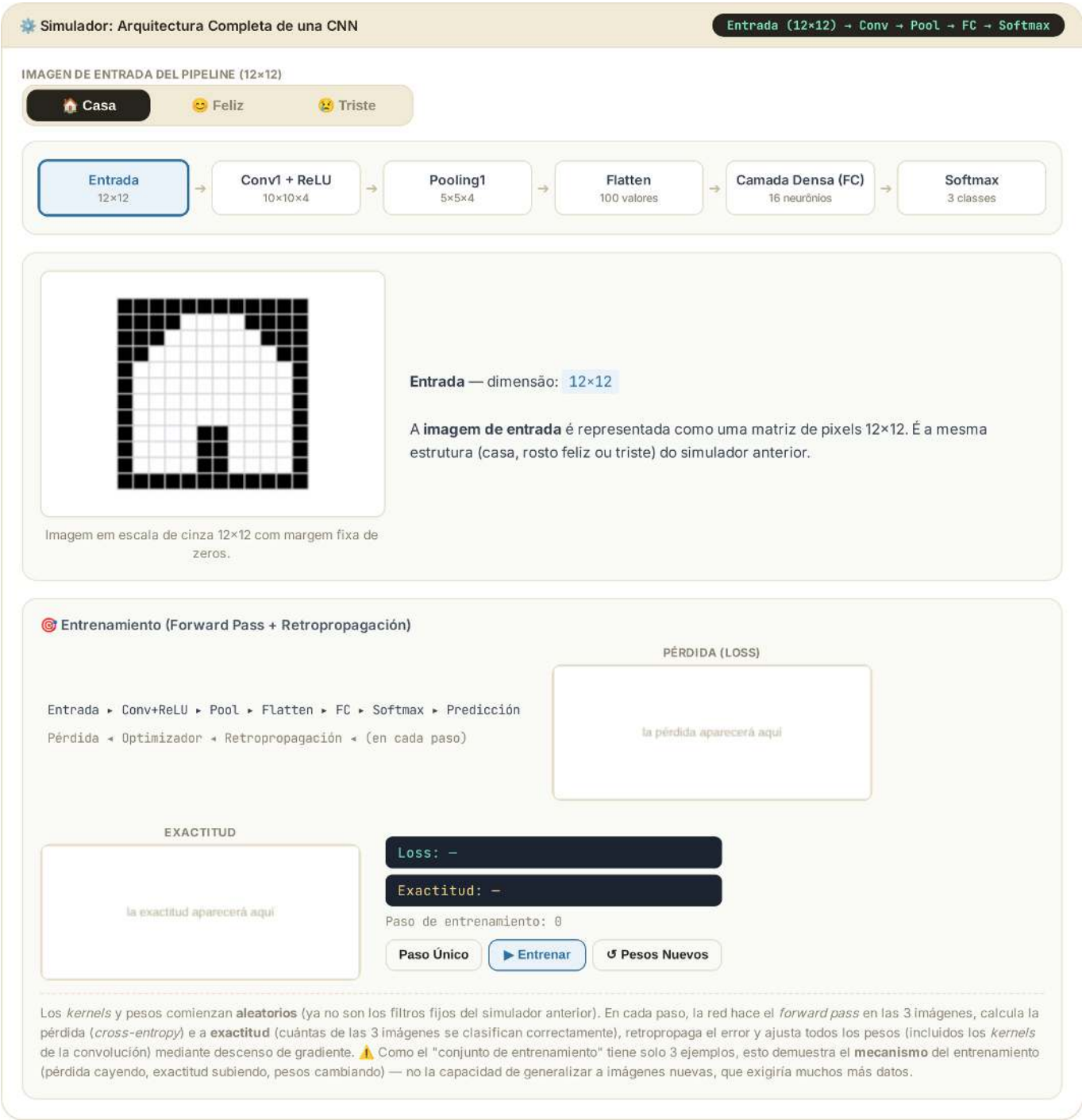
Sea $Z[r][c]$ el mapa de características preactivación en la posición (r, c) de la ventana deslizante, obtenido mediante la correlación cruzada entre el *kernel* K y la entrada X:

$$Z[r][c] = \sum_{k_r} \sum_{k_c} K[k_r][k_c] \cdot X[r + k_r][c + k_c]$$

Aplicando la regla de la cadena para determinar la contribución de un peso específico $K[k_r][k_c]$ en la función de pérdida L, se obtienen las siguientes etapas:

1. **Error Local (dZ):** En cada posición (r, c), se calcula la derivada parcial de la función de pérdida con respecto a la preactivación:
$$dZ[r][c] = \frac{\partial L}{\partial Z[r][c]}$$
que cuantifica la responsabilidad de esa posición específica en el error total de la red (L).

2. **Contribución del Peso:** Como $\frac{\partial Z[r][c]}{\partial K[k_r][k_c]} = X[r + k_r][c + k_c]$, la influencia de un peso específico $K[k_r][k_c]$ en el error de la posición (r, c) se obtiene multiplicando el error local $dZ[r][c]$ por el valor del píxel de entrada alineado con ese peso en el instante del cálculo:
$$dZ[r][c] \cdot X[r + k_r][c + k_c]$$



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-09-arquitetura.html

Figura 9.11: Simulador interactivo de la arquitectura de una CNN: elige una de las imágenes de entrada 12×12 (casa, rostro feliz o triste), haz clic en cada bloque del *pipeline* — Entrada, *Conv+ReLU*, *Pooling*, *Flatten*, *FC* y *Softmax* — y ejecuta pasos de entrenamiento reales (*forward pass* + retropropagación) para observar la pérdida y la precisión evolucionando, los *kernels* siendo ajustados y el *Softmax* empezando a apuntar a la clase correcta.

3. **Acumulación de Gradientes:** El gradiente final del peso corresponde a la **suma de las contribuciones ("votos") de todas las posiciones** recorridas por la ventana deslizante:

$$\frac{\partial L}{\partial K[k_r][k_c]} = \sum_{(r,c)} dZ[r][c] \cdot X[r+k_r][c+k_c]$$

Esta formulación asegura paridad directa entre la derivación analítica y los valores computados en el simulador de inspección del gradiente (Figura 9.12).

9.4.8.2 El Papel de la Función ReLU como "Filtro de Relevancia"

La aplicación de la función de activación **ReLU** ($\max(0, z)$) inmediatamente después de la convolución introduce una propiedad de selectividad al gradiente:

- **Activación Positiva ($Z[r][c] > 0$):** La derivada de la ReLU es 1. El error local se propaga integralmente ($dZ \neq 0$), permitiendo que la posición contribuya a la actualización de los pesos del *kernel*.
- **Activación Inactiva ($Z[r][c] \leq 0$):** La derivada de la ReLU es 0. El error local se anula ($dZ = 0$), suprimiendo la contribución de la posición al gradiente final.

  Nota didáctica: La ReLU asegura que solo las regiones espaciales que produjeron respuestas activas durante la propagación hacia adelante posean capacidad de modificar los pesos del *kernel* en el proceso de retropropagación.

9.4.8.3 Actualización de los Pesos mediante Descenso de Gradiente

Tras la consolidación de los gradientes acumulados de todas las posiciones, la actualización del peso ocurre según el algoritmo del Descenso de Gradiente Estocástico (SGD):

$$K[k_r][k_c] \leftarrow K[k_r][k_c] - \eta \cdot \frac{\partial L}{\partial K[k_r][k_c]}$$

donde η denota la **tasa de aprendizaje (*learning rate*)**.

- Si la suma de los gradientes es **positiva**, el valor del peso se reduce.
- Si la suma es **negativa**, el valor del peso se incrementa.

9.4.8.4 Explorando el Simulador Interactivo

> 🔗 De la Arquitectura Global a la Inspección del Gradiente
>
> En el simulador de arquitectura (Figura 9.11), se observa el error dZ derivado de la retropropagación multicapa completa, originado en la pérdida de entropía cruzada (*Softmax*) sobre las imágenes de entrada 12×12.
> Para viabilizar la verificación analítica del gradiente sin la sobrecarga de 100 posiciones de convolución y retropropagación multicapa, el simulador de aprendizaje del *kernel* (Figura 9.12) adopta un modelo de inspección reducido (6×6). En ese escenario, se simplifica el problema sustituyendo la clasificación compleja por un **metadato de calibración escalar**: se ajusta el filtro para producir una respuesta acumulada predefinida (alvo $= 9$) al identificar un patrón específico (como un borde a 45 grados). El mecanismo de acumulación de gradientes ($dZ \cdot X$) permanece rigurosamente idéntico en ambas formulaciones.

Para inspeccionar esta dinámica a nivel numérico, se utiliza el simulador en la Figura 9.12:

- **Imagen de Entrada (X):** Matriz 6×6.
- **Filtro Convolucional (K):** Matriz 3×3 (9 pesos).
- **Mapa de Salida (Z / A):** Matriz 4×4 (16 posiciones de la ventana).

- **Función de Pérdida (L):** Definida por $L = \frac{1}{2}(S - \text{alvo})^2$, donde $S = \sum A[r][c]$ representa la suma global de las activaciones post-ReLU.

  El papel del alvo $= 9$: El valor escalar alvo $= 9$ representa la "energía de activación" ideal estipulada para la imagen con borde diagonal. Como el mapa A posee 16 posiciones, ese valor equivale a buscar una respuesta media de $\frac{9}{16} \approx 0,56$ por píxel activado. Cuando $S > 9$, la red identifica que el filtro está reaccionando con intensidad excesiva al patrón, generando un error $dZ > 0$ que fuerza la reducción de los pesos K. Cuando $S < 9$, los pesos se incrementan para amplificar la señal.

Ruta Sugerida de Experimentación:

1. **Selección de Peso:** En la cuadrícula 3×3, elija el peso a analizar (ej.: $K[0][0]$).
2. **Barrido de la Ventana:** Utilice el botón **"▷ Avanzar posición"** para acompañar el desplazamiento de la ventana por las 16 posiciones espaciales. Note el resaltado visual en la celda del mapa de entrada que alinea el píxel X con el peso seleccionado.
3. **Análisis del Voto Local:** Examine el producto del error local por el píxel de entrada ($dZ \cdot X$) en el panel de cálculo de la posición.
4. **Verificación del Historial:** Acompañe la consolidación de los 16 resultados parciales organizados en las cuatro columnas de historial, observando la acumulación del gradiente final.
5. **Actualización del Kernel:** Haga clic en **"▷ Aplicar paso de descenso de gradiente"** para visualizar la convergencia de la curva de pérdida y la adaptación del *kernel* aleatorio al patrón de entrada seleccionado.

ℹ 🧠 Síntesis — De la convolución al aprendizaje de representaciones

Los simuladores de esta sección demuestran, de forma secuencial, cómo una CNN transforma una imagen de entrada en una estimación probabilística y cómo se optimizan sus parámetros durante el entrenamiento:

- **Convolución:** aplica filtros sobre la imagen para extraer características locales, generando mapas de características mediante el **uso compartido de pesos**.
- **ReLU:** introduce no linealidad al sistema, permitiendo el modelado de relaciones complejas entre los datos.
- ***Pooling*:** reduce la resolución espacial de los mapas de características, disminuyendo el costo computacional y otorgando invariancia a pequeñas traslaciones locales.
- ***Flatten*:** reorganiza los mapas multidimensionales en un vector unidimensional para alimentar las capas subsiguientes.
- **Capa totalmente conectada:** combina las características extraídas para producir las puntuaciones brutas (*logits*) asociadas a cada clase.
- **Softmax:** convierte los *logits* en una distribución de probabilidades normalizada.
- **Función de pérdida:** compara la distribución prevista con la verdad de referencia (*ground truth*), cuantificando escalarmente el error de la red.
- **Retropropagación:** aplica la regla de la cadena para calcular la derivada parcial (gradiente) de la función de pérdida con respecto a cada parámetro entrenable.
- **Optimizador:** actualiza los coeficientes de los filtros, pesos y sesgos en la dirección opuesta al gradiente, reduciendo la pérdida en cada iteración.

A lo largo de las iteraciones, los filtros convolucionales se convierten de valores estocásticos en detectores especializados: las capas iniciales aprenden primitivas visuales de bajo nivel (como bordes y texturas), mientras que las capas más profundas consolidan estas representaciones en estructuras abstractas y semánticas.

Simulador: Gradiente de un Peso del Kernel

∂Pérdida / ∂K[kr][kc] = Σ dZ · X

PATRÓN DE ENTRADA (IMAGEN 6×6)

Borde diagonal | Borde vertical

TASA DE APRENDIZAJE (η)

0.0005 (Lenta) | 0.002 (Ideal) | 0.02 (Alta)

Ejemplo **reducido**: imagen 6×6 y filtro 3×3 generando mapas 4×4. Haz clic en las pestañas "🔍 **¿Cómo se calcula?**" debajo de cada matriz para entender los cálculos paso a paso. **Pasa el mouse sobre cualquier celda** de X, Z, A, dZ o K para ver el cálculo exacto de ese valor, con los elementos usados en las capas relacionadas resaltados con contorno discontinuo/azul.

1. PESO DEL KERNEL

K[0][0]	K[0][1]	K[0][2]
K[1][0]	K[1][1]	K[1][2]
K[2][0]	K[2][1]	K[2][2]

KERNEL ACTUAL (K)

-0.34	0.13	-0.20
0.20	0.09	-0.05
0.32	-0.26	0.01

▸ 🔍 ¿Cómo se actualiza?

ENTRADA X (6×6)

1	0	0	0	0	0
1	1	0	0	0	0
1	1	1	0	0	0
1	1	1	1	0	0
1	1	1	1	1	0
1	1	1	1	1	1

▸ 🔍 ¿Cómo funciona X?

PRE-ACTIVACIÓN Z (4×4)

0.02	0.26	0.32	0.00
0.10	0.02	0.26	0.32
-0.10	0.10	0.02	0.26
-0.10	-0.10	0.10	0.02

▸ 🔍 ¿Cómo calcula Z?

ACTIVACIÓN A (4×4)

0.02	0.26	0.32	0.00
0.10	0.02	0.26	0.32
0.00	0.10	0.02	0.26
0.00	0.00	0.10	0.02

▸ 🔍 ¿Cómo calcula A?

ERROR dZ (4×4)

-7.20	-7.20	-7.20	0.00
-7.20	-7.20	-7.20	-7.20
0.00	-7.20	-7.20	-7.20
0.00	0.00	-7.20	-7.20

S = 1.800 (alvo = 9)

L = ½·(1.800 - 9)² = 25.9200

▸ 🔍 ¿Cómo calcula dZ, S y Loss?

2. CÁLCULO Y SUMA DE LOS "VOTOS" DE CADA POSICIÓN

∂L/∂K[0][0] = $\Sigma_{(r,c)}$ dZ[r][c] · X[r+0][c+0]

⏮ Volver posición | ▶ Avanzar posición | Sumar todo | ↺ Reiniciar posiciones

CÁLCULO DE ESTA POSICIÓN

Clique em "Avançar posição" para ver a primeira conta.

SUMA ACUMULADA (GRADIENTE)

posições: 0 / 16

total = 0.000

HISTORIAL DE LAS 16 POSICIONES (COLUMNAS c=0, c=1, c=2, c=3)

p(0,0) -7.20×1=-7.20	p(0,1) -7.20×0=0.00	p(0,2) -7.20×0=0.00	p(0,3) 0.00×0=0.00
p(1,0) -7.20×1=-7.20	p(1,1) -7.20×1=-7.20	p(1,2) -7.20×0=0.00	p(1,3) -7.20×0=0.00
p(2,0) 0.00×1=0.00	p(2,1) -7.20×1=-7.20	p(2,2) -7.20×1=-7.20	p(2,3) -7.20×0=0.00
p(3,0) 0.00×1=0.00	p(3,1) 0.00×1=0.00	p(3,2) -7.20×1=-7.20	p(3,3) -7.20×1=-7.20

3. USA EL GRADIENTE PARA ACTUALIZAR EL KERNEL

▶ Aplicar paso de descenso de gradiente | ⏮ Deshacer paso | 🎲 Nuevo kernel aleatorio

perda aparecerá aqui

Actualizaciones: 0

PÉRDIDA A LO LARGO DE LAS ACTUALIZACIONES

Estado Inicial do peso K[0][0] = -0.340:

- **Gradiente visível acumulado (∇K):** -50.400 (soma dos 16 votos dZ × X)
- **Passo de ajuste previsto (η × ∇K):** 0.002 × (-50.400) = **-0.1008**
- **Novo Peso previsto:** K ← -0.340 − (-0.1008) = **-0.239**

Clique em "**▶ Aplicar passo de gradiente descendente**" para atualizar a matriz.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-09-gradiente-kernel.html

Figura 9.12: Simulador interactivo del cálculo del gradiente de un peso del *kernel* convolucional.

9.5 Aplicaciones Prácticas en VC

Tras la consolidación teórica de los fundamentos de las CNNs y la verificación visual de cada una de sus operaciones elementales mediante los simuladores interactivos, resulta esencial observar la integración de estas etapas en *pipelines* completos de programación.

En las secciones siguientes, la teoría se traduce en código ejecutable en **PyTorch**, explorando las tres tareas fundamentales de la VC: **clasificación**, **detección de objetos** y **segmentación semántica**. Esta progresión práctica permite analizar desde la construcción de una arquitectura convolucional entrenada desde cero hasta la aplicación de estrategias avanzadas de **transferencia de aprendizaje** (*transfer learning*) en modelos preentrenados para conjuntos de datos sintéticos y reales.

9.5.1 Clasificación de Imágenes con CNNs

La clasificación de imágenes es una de las aplicaciones más tradicionales de las CNNs. En esta tarea, el objetivo es asignar una única etiqueta a la imagen de entrada, como identificar una categoría de objeto, una especie de animal o una clase de diagnóstico. Para ello, la CNN transforma progresivamente los valores de los píxeles en representaciones de mayor nivel de abstracción, combinando capas convolucionales, funciones de activación y operaciones de reducción espacial hasta producir una distribución de probabilidades entre las clases posibles. En esta sección, se presentan la arquitectura básica de una CNN clasificadora, el flujo de transformación de los datos a lo largo de la red y el proceso de entrenamiento para el ajuste de los parámetros aprendidos.

9.5.1.1 Entrenamiento de una CNN desde Cero en Dígitos

Para establecer una comparación directa con los enfoques presentados en el Capítulo 7, se desarrolla en esta sección una CNN entrenada sobre el mismo conjunto de datos de dígitos manuscritos (`load_digits`). La diferencia fundamental reside en la etapa de representación: mientras que los métodos clásicos dependen de *píxeles* brutos o de descriptores calculados manualmente, como el *Histogram of Oriented Gradients* (*HOG*), la CNN aprende automáticamente los coeficientes de los filtros convolucionales durante el proceso de optimización.

Los códigos siguientes (consolidados en la Figura 9.15) realizan la preparación de los datos, definen una arquitectura convolucional simple en **PyTorch**, ejecutan el bucle de entrenamiento mediante el algoritmo *Adam* y generan las curvas de evolución de la función de pérdida y de la precisión.

Bloque 1: Preparación y Estructuración de los Datos

La etapa inicial de cualquier *pipeline* de Aprendizaje Profundo consiste en la conversión y adecuación de los datos de entrada al formato exigido por el *framework* de computación científica.

El Concepto de *Tensor*

En Aprendizaje Profundo, la estructura fundamental de datos es el ***tensor***. Desde el punto de vista computacional, un *tensor* consiste en un arreglo multidimensional de números generalizado para n dimensiones:

- Un *tensor* de orden 0 es un escalar (un único valor).
- Un *tensor* de orden 1 es un vector (longitud).
- Un *tensor* de orden 2 es una matriz (filas y columnas).
- Un *tensor* de orden 3 o superior representa un volumen o hiper-arreglo de datos.

En el contexto de **PyTorch**, la clase `torch.Tensor` extiende la funcionalidad de los arreglos numéricos multidimensionales (como los de **NumPy**) al ofrecer soporte para operaciones aceleradas en hardware mediante *GPUs* (*Graphics Processing Units*) y soporte para el cálculo automático de derivadas (*autograd*), esencial para el algoritmo de retropropagación.

Análisis del Código de Preprocesamiento

1. **Carga y Normalización de las Intensidades:** El conjunto `load_digits` posee 1.797 muestras de dígitos manuscritos de 8×8 *píxeles*, cuyas intensidades originales varían en la escala entera de 0 a 16. La división por 16.0 realiza la **normalización** de los datos al rango $[0.0, 1.0]$. Esta transformación a escala flotante (`float32`) es indispensable en redes neuronales para evitar la saturación de las funciones de activación y estabilizar el cálculo de los gradientes en el algoritmo de optimización.

2. **División Estratificada (70% Entrenamiento / 30% Prueba):** La función `train_test_split` separa el 70% de las muestras para el ajuste de los parámetros de la red y reserva el 30% para la evaluación del modelo en datos no vistos. El parámetro `stratify=y` garantiza el muestreo estratificado, manteniendo la proporción exacta de cada una de las 10 clases de dígitos (0 a 9) en ambos conjuntos, previniendo sesgos de distribución.

3. **Adecuación Dimensional para Convolución 2D (`unsqueeze`):** En las CNN, las capas convolucionales bidimensionales (`nn.Conv2d`) exigen que el *tensor* de entrada posea estrictamente 4 dimensiones en el orden (N, C, H, W):

   - N: número de muestras (*batch size*).
   - C: número de canales de color (1 para escala de grises, 3 para *RGB*).
   - H: altura de la imagen en *píxeles* (8).
   - W: ancho de la imagen en *píxeles* (8).

   Como el arreglo original posee un formato 3D del tipo $(N, 8, 8)$, la llamada `.unsqueeze(1)` inserta una dimensión unitaria específicamente en el **índice 1** (la posición reservada para el canal de color C), transformando la estructura en un *tensor* 4D de formato $(N, 1, 8, 8)$, conforme lo exige PyTorch.

4. **Conversión de las Etiquetas (`dtype=torch.long`):** Las etiquetas de las clases y se convierten en *tensors* enteros de 64 *bits* (`torch.long`). Esta especificación de tipo es un requisito de la función de pérdida de Entropía Cruzada (`nn.CrossEntropyLoss`), que utiliza enteros no negativos como índices para asociar la clase correcta a los *logits* de salida de la red.

> **Atención a las Dimensiones:** La estructura final se representa mediante el **tensor `(N, 1, 8, 8)`**, donde `N` es el número de muestras (*batch size*), `1` es el canal de color (escala de grises) y `8×8` es la resolución espacial de la imagen en *píxeles*.

La Figura 9.13 ilustra una secuencia de muestras del conjunto de entrenamiento después del preprocesamiento y la adecuación dimensional a los *tensors* de **PyTorch**. En la etapa de visualización, la llamada `img.squeeze().numpy()` encadena dos transformaciones: el método `.squeeze()` elimina la dimensión unitaria redundante del canal de color, reduciendo el *tensor* 3D de formato `(1, 8, 8)` a una matriz 2D de `(8, 8)`; a continuación, el método `.numpy()` convierte la estructura de **PyTorch** en una matriz nativa de **NumPy**, formato exigido por las herramientas de renderizado gráfico como `mm.show()`.

```
# 1. Carga y preprocesamiento de los datos
digits = load_digits()
X = digits.images.astype(np.float32) / 16.0  # Normalización al rango [0, 1]
y = digits.target

# División estratificada en conjuntos de entrenamiento (70%) y prueba (30%)
X_treino, X_teste, y_treino, y_teste = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Adecuación a la dimensión esperada por PyTorch: (N_muestras, Canales, Altura, Anchura)
X_treino_t = torch.tensor(X_treino).unsqueeze(1)   # Dimensión: (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)
```

```
# Mostrar una muestra
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))
```

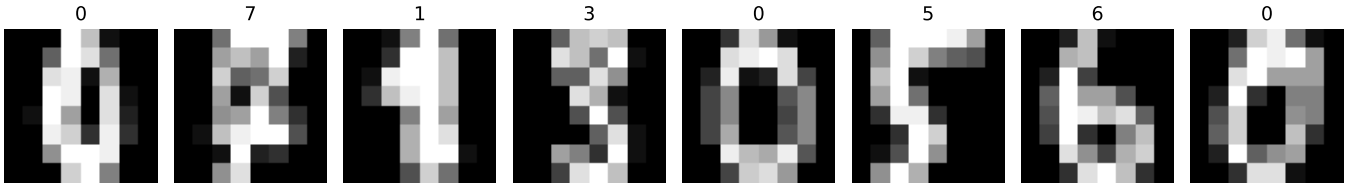


Figura 9.13: Muestras de dígitos del conjunto de entrenamiento tras conversión a *tensores* PyTorch y normalización.

Bloque 2: Definición de la Arquitectura Convolucional

La construcción de modelos en **PyTorch** se estructura a partir del paradigma de orientación a objetos, creándose una clase específica para representar la red neuronal (en este ejemplo, la clase `CNNDigitos`), la cual hereda todas las funcionalidades de la clase base `nn.Module`. El constructor `__init__` es responsable de instanciar las capas y declarar sus parámetros entrenables, mientras que el método `forward` establece la secuencia numérica de la propagación hacia adelante (*forward pass*).

Figura 9.14 sintetiza las transformaciones espaciales de los *tensores* y el flujo de datos a lo largo de la clase `CNNDigitos`.

Figura 9.14: Representación del flujo de transformaciones dimensionales de los *tensores* a lo largo de la arquitectura *CNNDigitos*.

1. **Constructor (`__init__`) e Instanciación de los Componentes:**
   - **Capa Convolucional 1 (`self.conv1`):** Aplica 8 filtros 3×3 con `padding=1` sobre la entrada en escala de grises (1 canal), preservando la resolución espacial de 8×8 *píxeles*.
   - **Capa Convolucional 2 (`self.conv2`):** Procesa los 8 mapas de características recibidos de la capa anterior aplicando 16 filtros 3×3 con `padding=1`.
   - **Submuestreo (`self.pool`):** Instancia la operación de *Max-Pooling* con ventana 2×2 y paso (*stride*) 2, reduciendo la dimensión espacial (altura y anchura) a la mitad en cada aplicación.
   - **Capas Totalmente Conectadas (`self.fc1` y `self.fc2`):** La primera proyección densa recibe el *tensor* aplanado de dimensión $16 \times 2 \times 2 = 64$ y produce 32 características intermedias. La segunda proyecta esas 32 características en los 10 *logits* finales de salida.
2. **Propagación hacia Adelante en el Método `forward`:**
   - **Primer Bloque Convolucional:** El *tensor* de entrada de formato $(N, 1, 8, 8)$ pasa por `conv1` + ReLU y es submuestreado por `pool`, resultando en el formato $(N, 8, 4, 4)$.
   - **Segundo Bloque Convolucional:** El *tensor* $(N, 8, 4, 4)$ es procesado por `conv2` + ReLU y reducido por `pool` al formato $(N, 16, 2, 2)$.
   - **Aplanamiento (*Flatten*):** El método `x.view(x.size(0), -1)` reconfigura la estructura 3D en un vector 1D de 64 elementos por muestra, preservando la dimensión del lote N.
   - **Clasificación:** El vector de 64 elementos alimenta `fc1` con activación ReLU (32 neuronas) y finaliza en `fc2`, produciendo los 10 *logits* no normalizados para el cálculo de la función de pérdida.

```python
# 2. Definición de la Arquitectura Convolucional
class CNNDigitos(nn.Module):
    """
    Arquitectura convolucional compacta:
    2 capas convolucionales con ReLU y Max-Pooling + 2 capas densas.
    """
    def __init__(self, n_classes=10):
        super().__init__()

        # Conv1: 1 canal de entrada, 8 filtros 3x3 con padding 1 (salida: 8x8)
        self.conv1 = nn.Conv2d(1, 8, kernel_size=3, padding=1)
        # Conv2: 8 canales de entrada, 16 filtros 3x3 con padding 1 (salida: 4x4)
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=1)

        self.relu = nn.ReLU()

        # Max-Pooling 2x2 con paso (stride) 2
        self.pool = nn.MaxPool2d(2, 2)

        # Capas totalmente conectadas (FC)
        self.fc1 = nn.Linear(16 * 2 * 2, 32)
        self.fc2 = nn.Linear(32, n_classes)

    def forward(self, x):
        # Primer bloque: Conv (8x8) -> ReLU -> Pool (4x4)
        x = self.pool(self.relu(self.conv1(x)))

        # Segundo bloque: Conv (4x4) -> ReLU -> Pool (2x2)
        x = self.pool(self.relu(self.conv2(x)))

        # Aplanamiento (Flatten): reconfigura la matriz 3D (16, 2, 2) en vector 1D (64)
        x = x.view(x.size(0), -1)

        # Capa densa intermedia con ReLU
        x = self.relu(self.fc1(x))

        # Capa final de clasificación (logits)
        return self.fc2(x)
```

Análisis de las Capas y del Flujo de la Clase `CNNDigitos`

1. **Constructor (`__init__`) e Instanciación de los Componentes:**
   - **Capa Convolucional 1 (`self.conv1`):** Aplica 8 filtros 3×3 con `padding=1` sobre la entrada en escala de grises (1 canal), preservando la resolución de 8×8 *píxeles*.
   - **Capa Convolucional 2 (`self.conv2`):** Procesa los 8 mapas de características recibidos aplicando 16 filtros 3×3 con `padding=1`.
   - **Submuestreo (`self.pool`):** Instancia la operación de *Max-Pooling* con ventana 2×2 y paso (*stride*) 2, reduciendo las dimensiones espaciales (alto y ancho) a la mitad en cada aplicación.
   - **Capas Totalmente Conectadas (`self.fc1` y `self.fc2`):** La primera proyección densa recibe el *tensor* aplanado de dimensión $16 \times 2 \times 2 = 64$ y produce 32 características intermediarias. La segunda proyecta esas 32 características en los 10 *logits* de salida.
2. **Propagación hacia Adelante en el Método `forward`:**
   - **Primer Bloque:** El *tensor* $(N, 1, 8, 8)$ pasa por `conv1` + ReLU y es reducido por `pool` a $(N, 8, 4, 4)$.
   - **Segundo Bloque:** El *tensor* $(N, 8, 4, 4)$ pasa por `conv2` + ReLU y es reducido por `pool` a $(N, 16, 2, 2)$.
   - **Aplanamiento (*Flatten*):** El método `x.view(x.size(0), -1)` convierte la estructura 3D en un vector 1D de 64 elementos por muestra.

- **Clasificación:** El vector de 64 elementos alimenta `fc1` con activación ReLU (32 neuronas) y finaliza en `fc2`, que produce los 10 *logits* finales para el cálculo de la pérdida de Entropía Cruzada.

Bloque 3: Instanciación y Parámetros de Optimización

La etapa de configuración del aprendizaje exige la instanciación de la arquitectura definida y la elección de dos componentes centrales: la **función de pérdida**, que cuantifica el error del modelo, y el **algoritmo de optimización**, responsable de ajustar los parámetros hacia el mínimo de dicha función.

1. **Instanciación y Conteo de Parámetros:** El modelo se crea a partir de la instanciación del objeto `modelo_cnn` de la clase `CNNDigitos`. La expresión `sum(p.numel() for p in modelo_cnn.parameters())` recorre todos los *tensores* de parámetros entrenables de la red (pesos y sesgos de cada capa) y calcula la cardinalidad total del modelo, cuantificando su capacidad de representación.

2. **Función de Pérdida (`nn.CrossEntropyLoss`):** La pérdida de Entropía Cruzada (*Cross-Entropy Loss*) es la elección estándar para problemas de clasificación multiclase. En PyTorch, esta implementación combina internamente la aplicación de la función *LogSoftmax* con la Pérdida de Log-Verosimilitud Negativa (*NLLLoss*). Por esta razón, la capa de salida de la red produce *logits* brutos, prescindiendo de la aplicación explícita de la función *Softmax* al final del método `forward`.

3. **Optimizador Adaptativo (`optim.Adam`):** La actualización de los parámetros utiliza el algoritmo *Adam* (*Adaptive Moment Estimation*), con una tasa de aprendizaje inicial $\eta = 0,01$ (`lr=1e-2`). El *Adam* combina los principios del momento con la adaptación del tamaño del paso basada en la media móvil de las derivadas de primer y segundo orden, ajustando individualmente la tasa de aprendizaje de cada parámetro de la red.

```
# 3. Inicialización del modelo y parámetros de optimización
modelo_cnn = CNNDigitos()
num_params = sum(p.numel() for p in modelo_cnn.parameters())
print(f"Parámetros entrenables del modelo: {num_params}")

criterio = nn.CrossEntropyLoss()
otimizador = optim.Adam(modelo_cnn.parameters(), lr=1e-2)
```

```
Parámetros entrenables del modelo: 3658
```

Bloque 4: Bucle de Entrenamiento y Evaluación

El entrenamiento de una CNN ocurre de forma iterativa mediante el algoritmo de Descenso de Gradiente Estocástico por *mini-lotes* (*Mini-batch SGD*).

Las curvas de aprendizaje resultantes de este proceso se presentan en la Figura 9.15, generada al final de la ejecución.

1. **Fase de Entrenamiento (`modelo_cnn.train()`):** El bucle principal ejecuta el entrenamiento a lo largo de 50 épocas. En cada época, ocurren las siguientes etapas:

   - **Barajado Estocástico:** La función `torch.randperm(n)` genera una permutación aleatoria de los índices de las muestras, garantizando que la ordenación de los *mini-lotes* varíe en cada época para evitar sesgos de muestreo.
   - **División en *Mini-lotes*:** El conjunto de entrenamiento se divide en lotes de 32 muestras (`tam_lote = 32`).
   - **Poner a cero los Gradientes (`otimizador.zero_grad()`):** Limpia los gradientes acumulados en el *tensor* en la iteración anterior, evitando la suma no deseada de derivadas entre lotes distintos.
   - **Paso hacia Adelante y Pérdida:** El *forward pass* calcula las predicciones `saida`, y la llamada `criterio(saida, y_treino_t[idx])` cuantifica el error del lote.

- **Retropropagación (`perda.backward()`):** Aplica la regla de la cadena para calcular las derivadas parciales de la pérdida con respecto a cada parámetro ($\frac{\partial L}{\partial w}$).
- **Actualización de los Pesos (`otimizador.step()`):** Actualiza los parámetros del modelo según las ecuaciones del optimizador *Adam*.

2. **Fase de Evaluación (`modelo_cnn.eval()`):** Al final de cada época, el modelo se cambia al modo de evaluación. El contexto `with torch.no_grad()` desactiva temporalmente el motor de cálculo automático de derivadas (*autograd*), reduciendo el consumo de memoria y acelerando la inferencia sobre el conjunto de prueba (`X_teste_t`). La operación `.argmax(dim=1)` extrae la clase de mayor probabilidad para cada muestra, permitiendo calcular la precisión de prueba.

3. **Visualización con la Biblioteca `morph`:** La función `mm.showTrainCurves` de la biblioteca didáctica `morph` consolida el historial de pérdida de entrenamiento y la precisión de prueba en un único panel gráfico, permitiendo diagnosticar la convergencia del modelo y monitorear la estabilidad del aprendizaje a lo largo de las épocas.

```
# 4. Bucle de Entrenamiento (Mini-batch SGD)
n = X_treino_t.size(0)                    # Número de muestras
tam_lote = 32                             # Tamaño del mini-lote
epocas = 50                               # Total de épocas
historico_perda, historico_acc = [], []   # Historial de métricas

for epoca in range(epocas):                  # Repite por época
    modelo_cnn.train()                       # Modo entrenamiento
    perm = torch.randperm(n)                 # Baraja muestras
    perda_epoca = 0.0                        # Acumula pérdidas
    for i in range(0, n, tam_lote):          # Recorre mini-lotes
        idx = perm[i:i + tam_lote]           # Índices del lote
        otimizador.zero_grad()               # Pon a cero gradientes
        saida = modelo_cnn(X_treino_t[idx])  # Propagación hacia adelante
        perda = criterio(saida, y_treino_t[idx]) # Calcula pérdida
        perda.backward()                         # Retropropagación
        otimizador.step()                        # Actualiza pesos
        perda_epoca += perda.item() * len(idx)   # Suma pérdida

    # Evaluación del modelo en el conjunto de prueba al final de cada época
    modelo_cnn.eval()                            # Modo evaluación
    with torch.no_grad():                        # Sin gradientes
        pred_teste = modelo_cnn(X_teste_t).argmax(dim=1)  # Predicciones
        acc_teste = (pred_teste == y_teste_t).float().mean().item()  # Exactitud
    historico_perda.append(perda_epoca / n)      # Registra pérdida
    historico_acc.append(acc_teste)              # Registra exactitud

acc_final_cnn = historico_acc[-1]                # Última exactitud
print(f"Exactitud final de la CNN en el conjunto de prueba: {acc_final_cnn:.4f}")  # Muestra
resultado

final = mm.showTrainCurves(                      # Traza curvas
    historico_perda, historico_acc,
    titulo="Evolução do Treinamento da CNN - Base de Dígitos",
    subtitulo=f"Acurácia final no teste: {acc_final_cnn:.4f}",
)
```

```
Exactitud final de la CNN en el conjunto de prueba: 0.9759
```

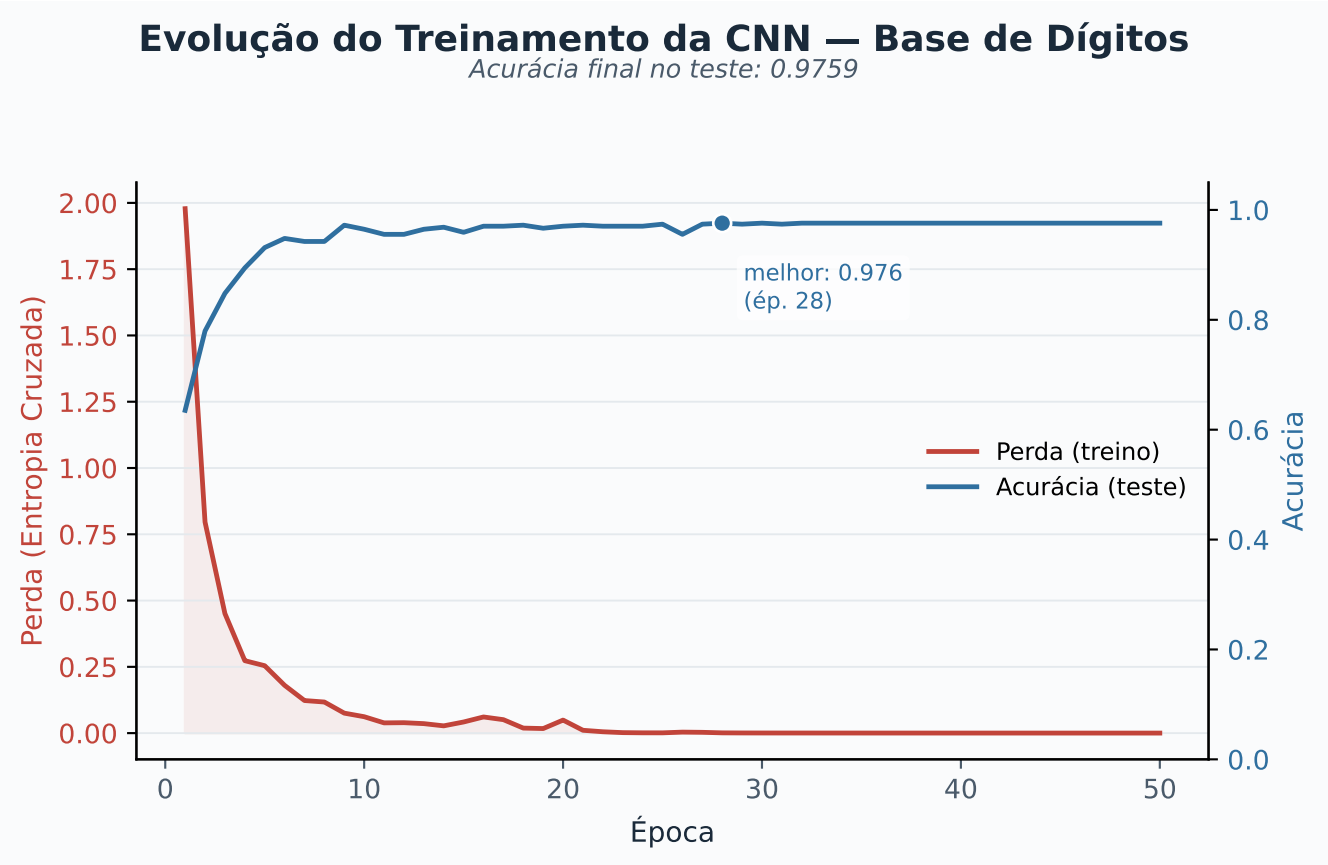


Figura 9.15: Curvas de entrenamiento y evaluación de la CNN en la base de dígitos: evolución de la pérdida de entropía cruzada en el conjunto de entrenamiento y de la exactitud en el conjunto de prueba a lo largo de 50 épocas.

Bloco 5: Visualización del Flujo de Activaciones

La inspección de la red entrenada permite observar la transformación progresiva del *tensor* de entrada a lo largo de las capas de la arquitectura `CNNDigitos`. La Figura 9.16 ilustra las dimensiones y las activaciones intermedias obtenidas al procesar un ejemplo real del dígito 3.

1. **Selección y Preparación de la Muestra:** La semilla estocástica se fija con `torch.manual_seed(7)` para asegurar la reproducibilidad de los resultados. La primera ocurrencia del dígito 3 en el conjunto de datos `load_digits` se aísla, se normaliza al intervalo $[0.0, 1.0]$ y se reconfigura como un *tensor* `x` de dimensión $(1, 1, 8, 8)$.

2. **Inspección Intermedia con `mm.showNet`:** La función `mm.showNet` de la biblioteca `morph` ejecuta la propagación hacia adelante (*forward pass*) del *tensor* `x` en la instancia `modelo_cnn` previamente entrenada. Utilizando *hooks* de *forward*, la función intercepta el estado numérico de las activaciones en las capas convolucionales (`nn.Conv2d`), de agrupamiento (`nn.MaxPool2d`) y totalmente conectadas (`nn.Linear`), devolviéndolas en el diccionario `acts`. Las funciones de activación no lineal (`nn.ReLU`) no se registran como etapas independientes, ya que su aplicación ocurre directamente sobre el *tensor* de salida de la capa correspondiente.

3. **Verificación de los Resultados:** La instrucción `list(acts.keys())` muestra la secuencia de identificadores de las capas monitoreadas, permitiendo confirmar la reducción dimensional progresiva y la generación del *logit* de valor máximo en el índice correspondiente a la clase 3, según se demuestra en la Figura 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)

# Reutilización de la instancia del modelo previamente entrenado
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("Capas capturadas:", list(acts.keys()))
```

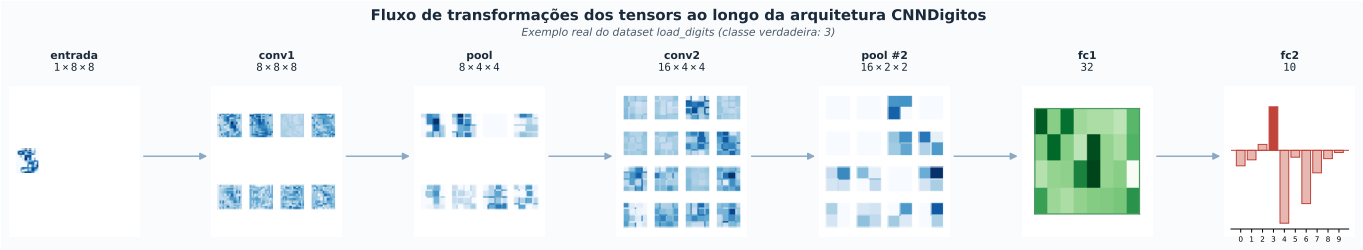


Figura 9.16: Flujo de activaciones de la CNN entrenada al procesar un ejemplo real del dígito 3, del *dataset load_digits*: dimensiones de los tensores capa a capa, de la entrada al *logit* de salida.

```
Capas capturadas: ['conv1', 'pool', 'conv2', 'pool #2', 'fc1', 'fc2']
```

Inspeccionando el Grafo Computacional con `torchviz`

Mientras que `mm.showNet` prioriza la claridad didáctica —mostrando una columna por capa con parámetros entrenables—, la biblioteca `torchviz` proyecta el **grafo de autograd** exactamente como PyTorch lo construye internamente para el cálculo de gradientes. La Figura 9.17 ilustra esta perspectiva al representar la arquitectura `CNNDigitos`.

1. **Propagación hacia adelante rastreada:** Con el modelo entrenado en modo `eval()`, el *forward pass* sobre el *tensor* `x` del dígito 3 es suficiente para que el motor de *autograd* registre todas las operaciones ejecutadas, incluyendo aquellas sin parámetros entrenables, como la función de activación `ReLU` y la reconfiguración dimensional `view`.

2. **Generación del grafo (`make_dot`):** La función `make_dot(saida, params=...)` construye el grafo a partir del *tensor* de salida, recorriendo retroactivamente el historial de operaciones hasta los nodos hoja (los parámetros entrenables del modelo). Cada nodo del diagrama representa una operación del *backward pass* (como `ReluBackward` o `AddmmBackward`), y no solo un bloque conceptual del `nn.Module`.

3. **Exportación y renderizado (`.render`):** El método `.render(..., format="png", cleanup=True)` invoca el ejecutable `dot` de **Graphviz** para compilar la imagen en formato PNG, eliminando automáticamente los archivos intermedios de código fuente.

La Figura 9.17 evidencia cómo este grafo computacional, incluso para una arquitectura compacta, presenta mayor densidad que el panel de `mm.showNet`, pues detalla cada operación atómica responsable del flujo de gradientes.

```
# 1. Forward pass con rastreo de gradiente habilitado
modelo_cnn.eval()
saida = modelo_cnn(x)  # Reutilización del tensor x (dígito 3)

# 2. Grafo básico: flujo de operaciones hasta la salida
```

```
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. Visualización directa en el entorno Quarto/Jupyter
# El .render() devuelve la ruta del archivo PNG generado; necesitamos abrirlo como imagen
imagem_simples = np.array(Image.open(caminho_simples).convert("RGB"))
mm.show(imagem_simples, figsize=(5,10))
```

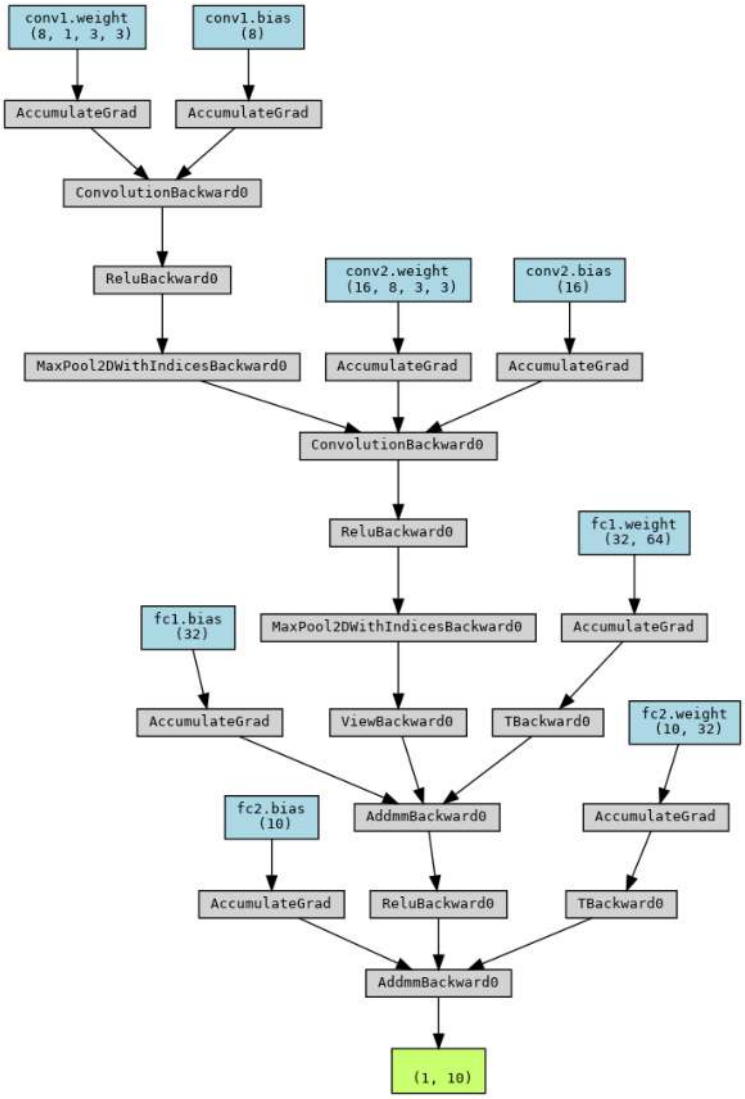


Figura 9.17: Grafo computacional de la *CNNDigitos* generado vía *torchviz*, mostrando las operaciones de *forward* y los nodos de gradiente (*backward*) asociados a cada parámetro entrenable.

Visão Detallada del Grafo Computacional con `torchviz`

Además de la representación simplificada, la biblioteca `torchviz` permite expandir el grafo de *autograd* para inspeccionar los detalles internos de ejecución de la red `CNNDigitos`. La Figura 9.18 presenta esta estructura expandida para el mismo *tensor* de entrada `x`.

1. **Rastreo con Atributos de Operación (`show_attrs=True`):** La inclusión de atributos muestra las configuraciones hiperparamétricas asociadas a cada nodo computacional durante la propagación hacia adelante (*forward pass*), tales como dimensiones de *kernel* (`kernel_size`), pasos (*stride*) y rellenos (*padding*) en las convoluciones y submuestreos.

2. **Detección de *Tensors* Guardados en Memoria (`show_saved=True`):** El parámetro fuerza la visualización explícita de los *tensors* intermedios que PyTorch retiene en memoria durante el *forward pass*. Estos datos se conservan porque serán estrictamente necesarios para el cálculo de las derivadas parciales durante la etapa de retropropagación (*backward pass*).

3. **Generación y Compilación de los Grafos:** Mientras que `grafo_simples` genera una vista directa del flujo de gradientes, `grafo_detalhado` compila el grafo expandido en el archivo `cnn_digitos_grafo_detalhado.png` mediante el ejecutable `dot` de **Graphviz**.

Como se observa en la Figura 9.18, esta visualización minuciosa es útil para depurar el consumo de memoria de video (*VRAM*) y verificar cómo el motor de PyTorch asigna internamente cada nodo de la regla de la cadena.

```
# mismo código anterior (make_dot + render)...

# 2. Grafo detallado: exhibición de dimensiones y tensores guardados para el backward
grafo_detalhado = make_dot(
    saida,
    params=dict(modelo_cnn.named_parameters()),
    show_attrs=True,   # Exhibe atributos de las operaciones (ej.: kernel_size, stride)
    show_saved=True,   # Exhibe tensores guardados en la memoria para la retropropagación
)

caminho_detalhado = grafo_detalhado.render("cnn_digitos_grafo_detalhado",
                                           format="png", cleanup=True)

# 3. Exhibición directa en el entorno Quarto/Jupyter
imagem_detalhado = np.array(Image.open(caminho_detalhado).convert("RGB"))
mm.show(imagem_detalhado, figsize=(8, 16))
```

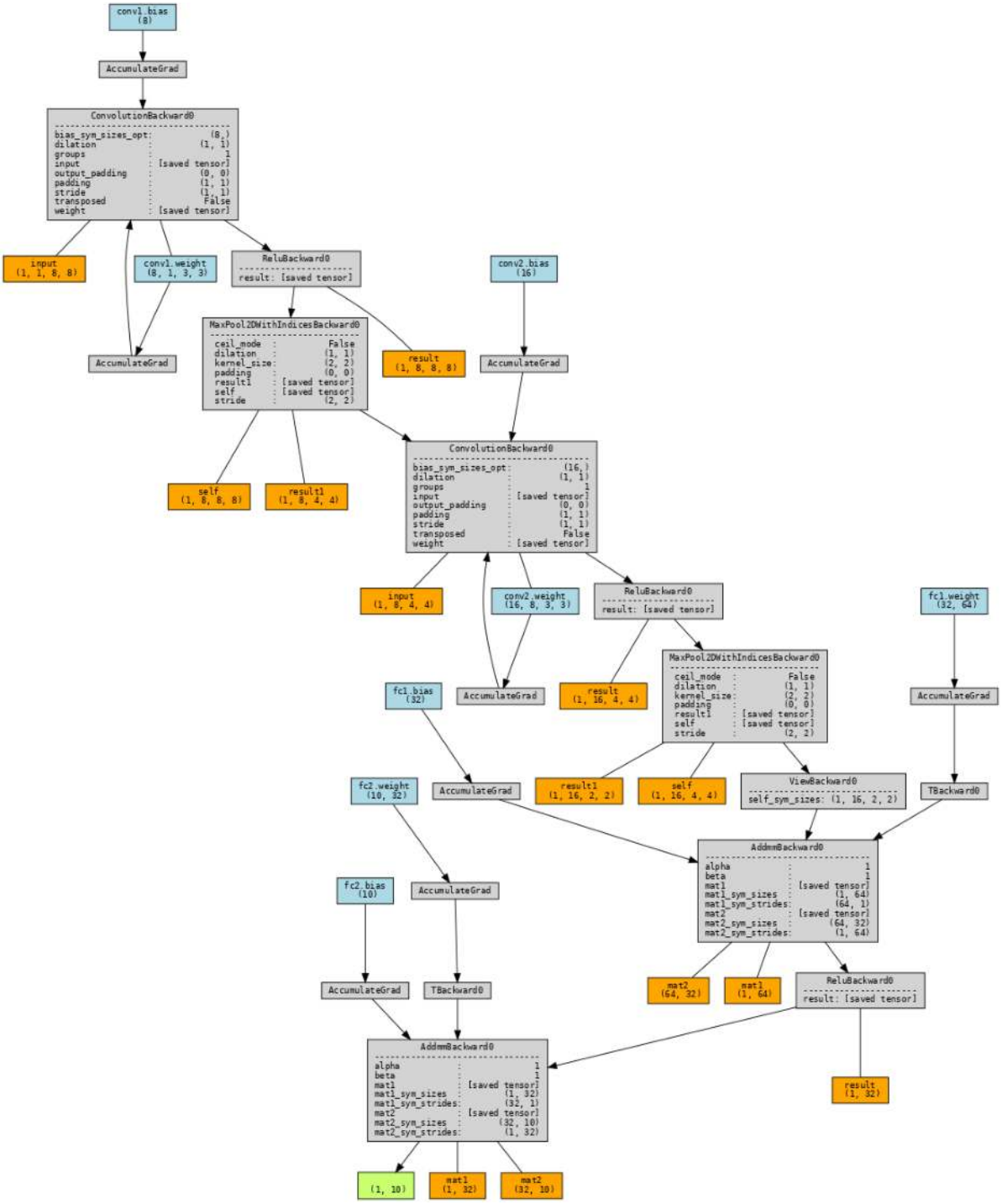


Figura 9.18: Grafo detallado de la clase CNNDigitos generado via *torchviz*.

Comparando con el Capítulo 7

La Figura 9.19 reúne los resultados obtenidos en el mismo conjunto de datos (`load_digits`), estableciendo un paralelo directo entre los enfoques clásicos explorados anteriormente y la CNN desarrollada en este capítulo.

1. **Rendimiento de los *píxeles* brutos vs. descriptores manuales:** En los experimentos del Capítulo 7, el clasificador k-NN ($k = 3$) alcanzó una precisión del $98,4\%$ cuando se alimentó directamente con los *píxeles* brutos de las imágenes. En contraste, la extracción previa de características mediante *Histogram of Oriented Gradients* (*HOG*) resultó en un rendimiento significativamente inferior ($75,8\%$). Esta caída se debe a que el *HOG* fue diseñado para capturar gradientes de bordes en imágenes de mayor resolución; en matrices de solo 8×8 *píxeles*, la resolución espacial es insuficiente para formar histogramas de

orientación informativos.

2. **Equivalencia de la CNN y aprendizaje *end-to-end*:** La red convolucional `CNNDigitos` alcanza un rendimiento competitivo del $97,6\%$, acercándose a la precisión del k-NN con *píxeles* brutos en una base pequeña y prealineada. La gran ventaja conceptual reside en el aprendizaje de representación: en lugar de depender de descriptores diseñados manualmente (*handcrafted features*) o de mantener todo el conjunto de datos en memoria para la búsqueda de vecinos en el momento de la inferencia, la CNN optimiza automáticamente sus propios filtros convolucionales durante el entrenamiento, generando un modelo compacto capaz de realizar la extracción de características y la clasificación de forma integrada (*end-to-end*).

```
import matplotlib.pyplot as plt

# Valores obtenidos en el Capítulo 7 (k-NN, k=3), reproducidos para comparación directa
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()
```

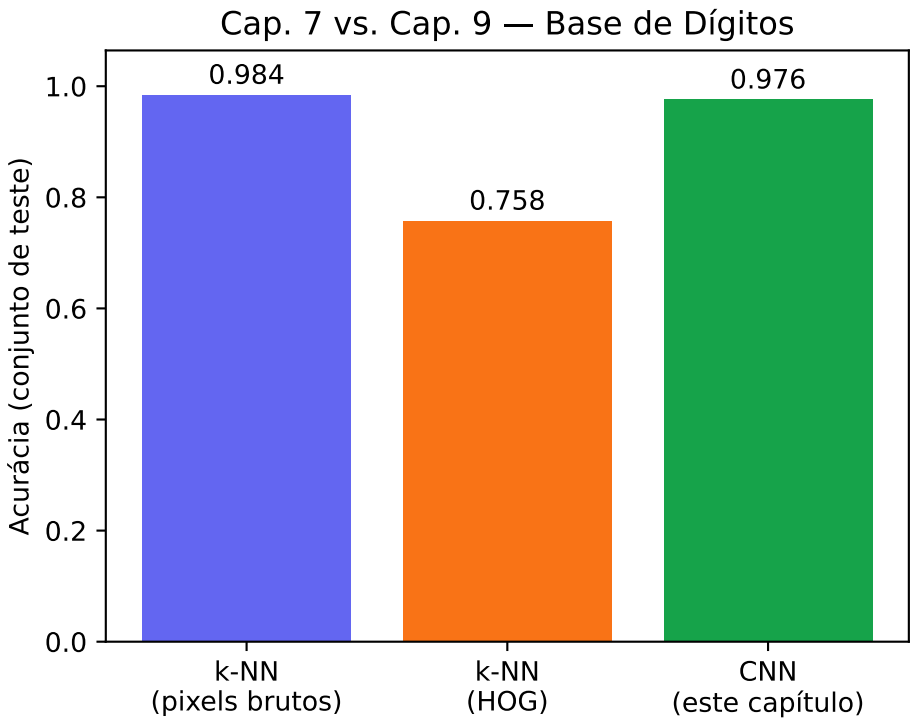


Figura 9.19: Comparación de precisión entre los clasificadores clásicos del Capítulo 7 (píxeles brutos y HOG con k-NN) y la CNN entrenada en este capítulo, en la misma base de dígitos.

¿Por qué funciona? — Y por qué la CNN no siempre "gana"

El resultado aquí observado **repite el patrón ya visto en el Capítulo 7**: la CNN, aunque aprende sus características automáticamente, no supera necesariamente al k-NN con *píxeles* brutos en esta base específica. La explicación es la misma: `load_digits` es una base pequeña (menos de 1.800 ejemplos), con imágenes ya centradas, normalizadas y de muy baja resolución (8×8) — condiciones en las que la comparación directa de intensidades ya es altamente informativa, y hay pocos datos para que la red aprenda filtros verdaderamente superiores a los descriptores simples.
El verdadero diferencial de las CNN aparece en escenarios que los descriptores artesanales y los clasificadores simples no pueden abordar: imágenes más grandes y realistas, con miles de categorías, variación sustancial de pose, iluminación y fondo, y conjuntos de entrenamiento masivos — exactamente el régimen en el que los modelos presentados en la sección *"Aplicaciones a Gran Escala"*, más adelante, fueron entrenados. La lección pedagógica que atraviesa los Capítulos 7, 8 y 9 de este libro es consistente: **la sofisticación de un método debe ser proporcional a la complejidad del problema** — usar una CNN para un problema que un k-NN resuelve igualmente bien es desperdicio de recursos computacionales, no una virtud.
Esa misma proporcionalidad se aplica a las herramientas de inspección utilizadas a lo largo del capítulo. El `mm.showNet` fue construido con fines **didácticos** y funciona bien en redes poco profundas como la `CNNDigitos`, pero no escala a arquitecturas profundas: cada capa rastreada se convierte en una columna en la figura, y las capas convolucionales con cientos de canales generan mosaicos demasiado grandes para la interpretación visual; además, los *hooks* almacenan todas las activaciones en memoria, y el *layout* asume un flujo secuencial, sin representar fielmente conexiones residuales o ramificaciones (como en *ResNets* o módulos *Inception*). Así, `showNet` debe entenderse como una lente pedagógica para redes pequeñas — análoga al papel de `mm.showBoundBox` en la depuración visual de detecciones — y no como un sustituto de herramientas orientadas a la producción, como *TensorBoard* o *torchviz*.

9.5.1.2 Transferencia de Aprendizaje

Entrenar una CNN desde cero generalmente requiere una gran cantidad de datos etiquetados y recursos computacionales significativos, ya que el proceso de entrenamiento necesita ajustar todos los parámetros de la red. En muchas aplicaciones, sin embargo, solo se dispone de un conjunto reducido de datos para la tarea de interés. En esta situación, la **transferencia de aprendizaje** (*transfer learning*) reutiliza las representaciones aprendidas por un modelo previamente entrenado en una tarea de origen con un gran volumen de datos, reduciendo el costo de entrenamiento y la necesidad de nuevas muestras.

En la VC, esta estrategia explora la organización jerárquica de las CNNs. Las capas iniciales aprenden características visuales de bajo nivel, como bordes, texturas, gradientes de intensidad y patrones de colores, que permanecen útiles en diferentes dominios. Las capas más profundas combinan esta información para formar representaciones progresivamente más abstractas y especializadas, relacionadas con las clases presentes en la base de entrenamiento.

Esta sección investiga en qué condiciones la transferencia de aprendizaje produce buenos resultados. El primer experimento muestra que un extractor pequeño y entrenado en un dominio restringido puede conducir a la **transferencia negativa** (*negative transfer*). El segundo demuestra por qué los modelos profundos preentrenados en grandes bases de imágenes alcanzan un alto rendimiento en nuevas tareas. Por último, el tercero aplica esta estrategia a un problema de diagnóstico fitosanitario, ilustrando un escenario cercano a aplicaciones reales.

Experimento 1 — Limitaciones de un Extractor Pequeño y Especializado

El primer experimento muestra que la transferencia de aprendizaje no siempre mejora el rendimiento de un modelo. Para ello, el conjunto de dígitos manuscritos (`load_digits`) se divide en dos dominios disjuntos:

- **Dominio A (origen):** dígitos 0 a 4, utilizados para entrenar una pequeña CNN;

- **Dominio B (destino):** dígitos 5 a 9, reindexados a 0 a 4, formando una nueva tarea con solo 20 muestras de entrenamiento.

El objetivo consiste en evaluar el efecto de reutilizar el extractor de características aprendido en el Dominio A sin permitir su adaptación al Dominio B.

Bloque 1: División de los Dominios, Escasez y Exhibición de las Muestras

Este bloque prepara el conjunto de datos para el experimento. A diferencia del proyecto anterior, que utilizó todos los dígitos en un único problema de clasificación, la base se divide en dos tareas independientes: una tarea de origen (Dominio A) y una tarea de destino (Dominio B).

La Figura 9.20 presenta ejemplos de los dos dominios después de la separación de las clases, la conversión a *tensors* de **PyTorch** y el preprocesamiento.

1. **Separación de las clases:** Las máscaras booleanas `mask_A` y `mask_B` separan los ejemplos de cada dominio. A continuación, el código reindexa las etiquetas del Dominio B (`y[mask_B] - 5`) al intervalo $[0, 4]$, permitiendo que ambos modelos utilicen cinco clases de salida.
2. **Escasez de datos:** El generador `np.random.default_rng(0)` selecciona solo 20 muestras para el entrenamiento del Dominio B, aproximadamente cuatro por clase, simulando un escenario en el que el entrenamiento desde cero tiende a sufrir de *overfitting*.
3. **Conversión a *tensors*:** La función `para_tensor` convierte las imágenes al formato $(N, 1, 8, 8)$ y las etiquetas a `torch.long`, compatibles con las capas `nn.Conv2d` y la función de pérdida.
4. **Visualización de las muestras:** El código utiliza `.squeeze().numpy()` para convertir los *tensors* en matrices de **NumPy**. La Figura 9.20 presenta ejemplos de los dos dominios y evidencia la reindexación aplicada a las etiquetas del Dominio B.

```
# 1. División del dataset en dos dominios disjuntos
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  # Reindexación de las etiquetas al intervalo [0, 4]

# División en entrenamiento y prueba para ambos dominios
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
)

# Simulación de escasez extrema en el dominio de destino: solo 20 muestras de entrenamiento
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]

# Función auxiliar para conversión a tensores PyTorch
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)

# Visualización de muestras de ambos dominios
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)
)
```

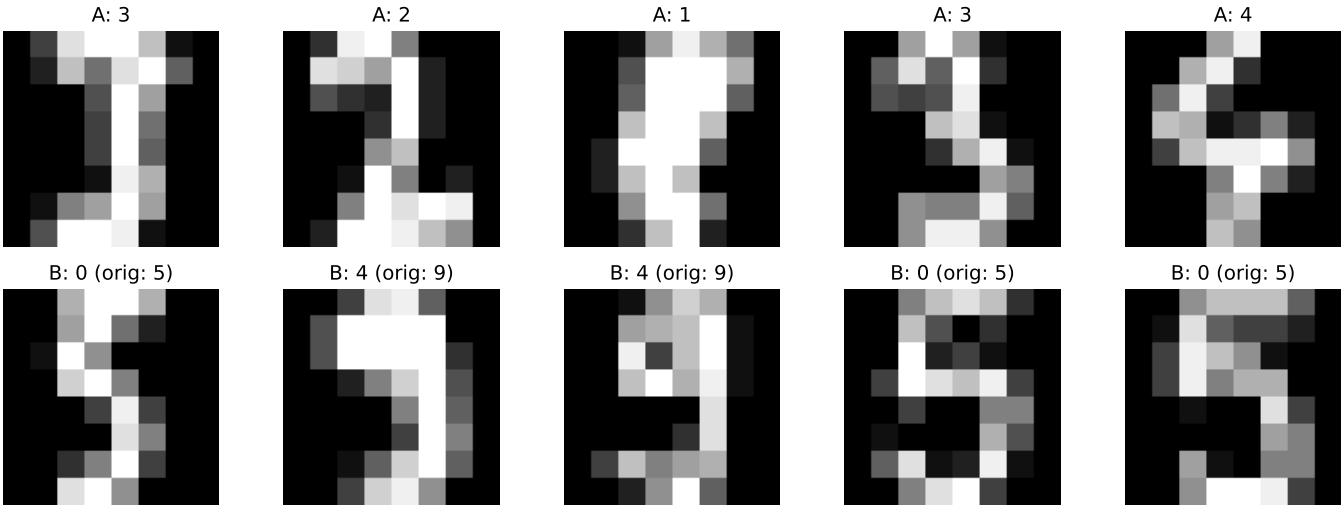


Figura 9.20: Muestras de los conjuntos de entrenamiento tras preprocesamiento y adecuación dimensional a los tensores PyTorch: Dominio A (dígitos 0 a 4, tarea de origen) y Dominio B (dígitos 5 a 9 reindexados a 0 a 4, tarea de destino).

Bloque 2: Arquitectura Modular y Rutinas Genéricas

Para viabilizar la transferencia de aprendizaje, la arquitectura convolucional y el bucle de entrenamiento fueron refactorizados en relación con la clase `CNNDigitos` del proyecto anterior.

1. **Modularización de la Arquitectura (Diferencia respecto a `CNNDigitos`):**
   - En el proyecto anterior, la clase `CNNDigitos` declaraba todas las capas (`conv1`, `conv2`, `pool`, `fc1`, `fc2`) como miembros directos de una única clase monolítica.
   - Aquí, la arquitectura se separa en dos componentes: la clase `ExtratorConv` encapsula el bloque espacial convolucional (2 convoluciones 3×3, 2 *Max-Poolings* 2×2 y el aplanamiento a 64 elementos), mientras que la clase `CNNCompleta` instancia ese extractor en `self.extrator` y anexa la "cabeza" clasificadora (`fc1` y `fc2`).
   - Esta separación es lo que permite copiar el estado interno del extractor (`state_dict()`) de un modelo a otro de forma aislada.

2. **Ajuste en el Número de Clases de Salida:** Mientras que `CNNDigitos` en el proyecto anterior poseía 10 *logits* en la capa de salida (`self.fc2 = nn.Linear(32, 10)`), la clase `CNNCompleta` recibe `n_classes=5` en el constructor para adecuarse a la división de los dominios A y B.

3. **Flexibilización del Bucle de Entrenamiento (`treinar`):**
   - En el proyecto anterior, el bucle de entrenamiento iteraba directamente sobre los atributos globales del modelo (`modelo_cnn.parameters()`) y calculaba métricas específicas en línea.

- La función `treinar` abstrae ese proceso e introduce el parámetro opcional `parametros`. Si se proporciona, el optimizador *Adam* actualiza **solo** los parámetros de esa lista, ignorando las capas cuyos gradientes fueron desactivados. Esta flexibilidad es crucial para ejecutar el entrenamiento con congelamiento parcial de la red.

4. **Aislamiento de la Evaluación (`calcular_acuracia`):** Así como se hizo en la fase de prueba del proyecto anterior, la función coloca el modelo en `eval()` y utiliza el contexto `torch.no_grad()` para desactivar el *autograd*, calculando la precisión mediante `.argmax(dim=1)`.

```
# Definición del bloque convolucional reutilizable (misma extracción del proyecto anterior)
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)

# Arquitectura modular que combina el extractor y la cabeza clasificadora
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)

# Rutina genérica de entrenamiento con optimización selectiva de parámetros
def treinar(modelo, X_t, y_t, epocas, lr, tam_lote=16, parametros=None):
    # Parámetros entrenables
    params = parametros if parametros is not None else modelo.parameters()
    otim = optim.Adam(params, lr=lr)                # Optimizador Adam
    crit = nn.CrossEntropyLoss()                    # Función de pérdida
    n_amostras = X_t.size(0)                        # Número de muestras
    for _ in range(epocas):                         # Repite por época
        perm = torch.randperm(n_amostras)           # Mezcla muestras
        for i in range(0, n_amostras, tam_lote):    # Recorre mini-lotes
            idx = perm[i:i + tam_lote]              # Índices del lote
            otim.zero_grad()                        # Pon a cero gradientes
            perda = crit(modelo(X_t[idx]), y_t[idx])  # Calcula pérdida
            perda.backward()                        # Retropropagación
            otim.step()                             # Actualiza pesos

# Rutina de evaluación
def calcular_acuracia(modelo, X_t, y_t):
    modelo.eval()                                   # Modo evaluación
    with torch.no_grad():                           # Sin gradientes
        pred = modelo(X_t).argmax(dim=1)            # Clases predichas
    return (pred == y_t).float().mean().item()      # Devuelve precisión
```

Bloco 3: Preentrenamiento, Transferencia y Análisis Comparativo

Este bloque ejecuta la comparación entre el ajuste del modelo desde cero y la aplicación de la transferencia con congelamiento estático del extractor. Para dar total transparencia al experimento, los tamaños de los conjuntos de entrenamiento y prueba de ambos dominios se imprimen en el terminal.

1. **Cuantificación de las Muestras por Dominio:**
   - **Dominio A (origen, dígitos 0 a 4):** Cuenta con 675 muestras de entrenamiento (75%) y 226 de prueba (25%), proporcionando datos abundantes para que el `modelo_origem` aprenda el extractor convolucional hasta alcanzar 100% de precisión.
   - **Dominio B (destino, dígitos 5 a 9):** Posee 224 muestras de prueba en total, pero su conjunto de entrenamiento se reduce intencionalmente de 672 a solo 20 **muestras** (`XB_tr_poucos`), creando un escenario severo de escasez de datos.
2. **Etapa 1: Preentrenamiento en el Dominio A (Origen):** El `modelo_origem` se entrena desde cero sobre las 675 muestras de los dígitos 0 a 4. Durante 40 épocas, el extractor convolucional ajusta sus filtros para identificar los rasgos característicos de esos cinco primeros dígitos, alcanzando 100% de precisión en el conjunto de prueba (226 muestras).
3. **Etapa 2: Transferencia de Pesos y Congelamiento:**
   - Se crea el `modelo_transferencia` para resolver la tarea del Dominio B (dígitos 5 a 9).
   - Los pesos aprendidos en el Dominio A se copian mediante:
     - `load_state_dict(modelo_origem.extrator.state_dict())`
   - **Congelamiento:** El bucle `for p in modelo_transferencia.extrator.parameters(): p.requires_grad = False` desactiva el cálculo de gradientes en las capas convolucionales.
   - **Entrenamiento Selectivo:** La llamada `treinar(...)` pasa estrictamente los parámetros de las capas densas (`params_cabeca`), ajustando la cabeza de clasificación con solo las 20 muestras de entrenamiento.
4. **Etapa 3: Entrenamiento desde Cero en el Dominio B (Control Experimental):** El `modelo_do_zero` posee la misma arquitectura, pero se entrena desde cero sobre las mismas 20 muestras del Dominio B, sin ningún reaprovechamiento de pesos, durante las mismas 40 épocas.
5. **Análisis de los Resultados (Figura 9.21):**
   - **Con Transferencia Congelada ($72,77\%$):** Al reaprovechar el extractor entrenado en el Dominio A y congelar sus parámetros, la red alcanza $72,77\%$ de precisión en la prueba (224 muestras) ajustando solo las capas densas.
   - **Entrenado desde Cero ($76,79\%$):** El entrenamiento desde cero supera a la transferencia congelada en el conjunto de prueba del Dominio B.
   - **Causa de la Diferencia:** Por tratarse de un modelo minúsculo (solo 16 filtros convolucionales en matrices de 8×8), el extractor entrenado en el Dominio A se volvió **hiperespecializado** en las formas geométricas de los dígitos 0 a 4. Al congelar rígidamente esos pocos filtros, el modelo de destino quedó limitado a detectores inadecuados para 5 a 9. La red entrenada desde cero, incluso con solo 20 muestras, logró adaptar sus 16 filtros directamente a los rasgos del Dominio B.

```
# Fixar semente para reprodutibilidade
torch.manual_seed(42)

# Exibição do tamanho dos grupos de treino e teste
print("=== Detalhamento do Tamanho das Bases ===")
print(f"Domínio A (0-4) - Treino: {len(XA_tr_t)} amostras | Teste: {len(XA_te_t)} amostras")
print(f"Domínio B (5-9) - Treino completo: {len(XB_tr)} | Treino reduzido:
{len(XB_tr_poucos)}",
      f"| Teste: {len(XB_te_t)} amostras\n")
```

```
# 1. Pré-treinamento na tarefa de origem (Domínio A: dígitos 0-4)
modelo_origem = CNNCompleta(n_classes=5)                        # Cria CNN

#######
treinar(modelo_origem, XA_tr_t, yA_tr_t, epocas=40, lr=1e-2)   # Treina modelo

acc_A = calcular_acuracia(modelo_origem, XA_te_t, yA_te_t)     # Mede acurácia

print(f"Acurácia no domínio de origem A "                       # Exibe resultado
      f"(dígitos 0-4, {len(XA_te_t)} testes): " f"{acc_A:.4f}")

# 2. Transferência de Aprendizado (Extrator Congelado)
modelo_transferencia = CNNCompleta(n_classes=5)        # Cria CNN
modelo_transferencia.extrator.load_state_dict(         # Copia extrator
    modelo_origem.extrator.state_dict())

for p in modelo_transferencia.extrator.parameters():   # Percorre extrator
    p.requires_grad = False                            # Congela pesos

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,              # Treina cabeça
         epocas=40, lr=1e-2, parametros=params_cabeca)

acc_transferencia = calcular_acuracia(modelo_transferencia, XB_te_t, yB_te_t) # Mede acurácia

# 3. Treinamento do Zero no Domínio B
modelo_do_zero = CNNCompleta(n_classes=5)                     # Cria CNN

#######
treinar(modelo_do_zero, XB_tr_t, yB_tr_t, epocas=40, lr=1e-2) # Treina modelo

acc_do_zero = calcular_acuracia(modelo_do_zero,  XB_te_t, yB_te_t) # Mede acurácia


print(f"Domínio de destino B (dígitos 5-9), apenas {len(XB_tr_poucos)} ",
      f"exemplos de treino ({len(XB_te_t)} testes):")
print(f"  Com transferência (extrator congelado): {acc_transferencia:.4f}")
print(f"  Treinando do zero (mesmos dados/épocas): {acc_do_zero:.4f}")

# Visualização comparativa
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()
```

```
=== Detalhamento do Tamanho das Bases ===
Domínio A (0-4) - Treino: 675 amostras | Teste: 226 amostras
Domínio B (5-9) - Treino completo: 672 | Treino reduzido: 20 | Teste: 224 amostras
```

```
Acurácia no domínio de origem A (dígitos 0-4, 226 testes): 1.0000
Domínio de destino B (dígitos 5-9), apenas 20  exemplos de treino (224 testes):
  Com transferência (extrator congelado): 0.7277
  Treinando do zero (mesmos dados/épocas): 0.7679
```

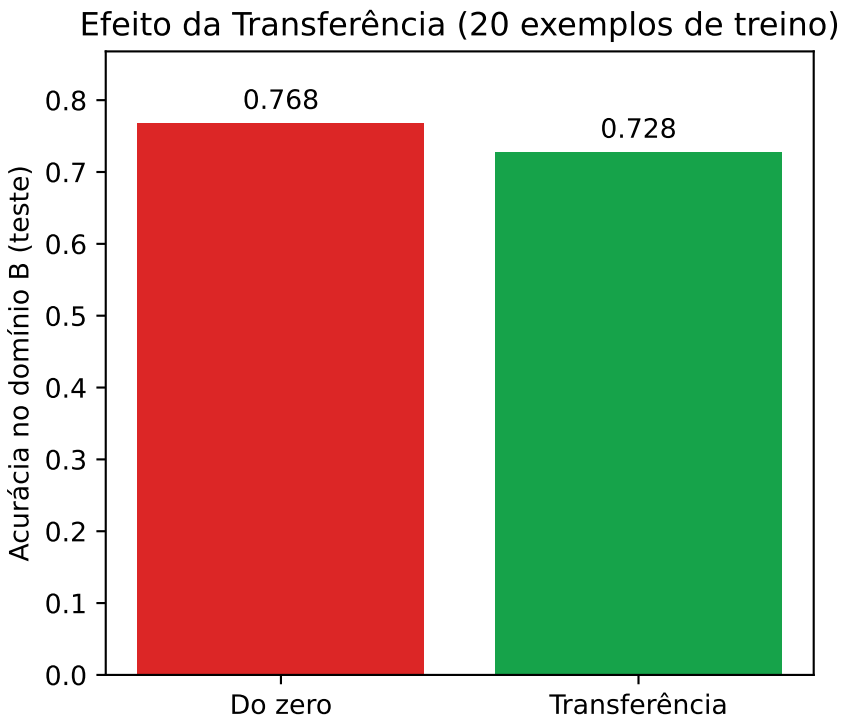


Figura 9.21: Comparação de acurácia no conjunto de teste do Domínio B (dígitos 5 a 9) sob restrição de dados (20 exemplos de treino): demonstração do impacto do congelamento rígido e da transferência negativa em redes de baixa capacidade.

Bloco 4: Visualización del Flujo de Activaciones con `mm.showNet`

Para confirmar que la extracción de características reutilizada preserva las transformaciones dimensionales estudiadas en el proyecto anterior, se utiliza nuevamente la función `mm.showNet` de la biblioteca `morph`. La Figura 9.22 muestra el flujo de activaciones del `modelo_transferencia` al procesar una muestra del Dominio B (dígito 7, reindexado a la clase 2).

1. **Preservación del Flujo Convolucional:** Dado que la arquitectura `ExtratorConv` replica las mismas capas de convolución y *pooling* de la `CNNDigitos` del proyecto anterior, las dimensiones de los *tensores* intermedios se mantienen en $(1, 8, 4, 4)$ en el primer bloque.
2. **Inspección de la Cabeza Adaptada:** La diferencia respecto al proyecto anterior surge en la capa de salida (`fc2`): mientras que el modelo del proyecto anterior proyectaba el vector intermedio en 10 *logits* (clases de 0 a 9), el modelo de transferencia proyecta el vector en 5 *logits* (clases de 0 a 4), capturando las probabilidades relativas del Dominio B.

```
# Selecciona la primera muestra de prueba del Dominio B
x_amostra_B = XB_te_t[0:1]  # Tensor de dimensión (1, 1, 8, 8)
classe_verdadeira = yB_te_t[0].item()
classe_original = classe_verdadeira + 5

# Inspección del flujo de activaciones en el modelo de transferencia
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("Capas capturadas en el modelo de transferencia:\n", list(acts_transfer.keys()))
```

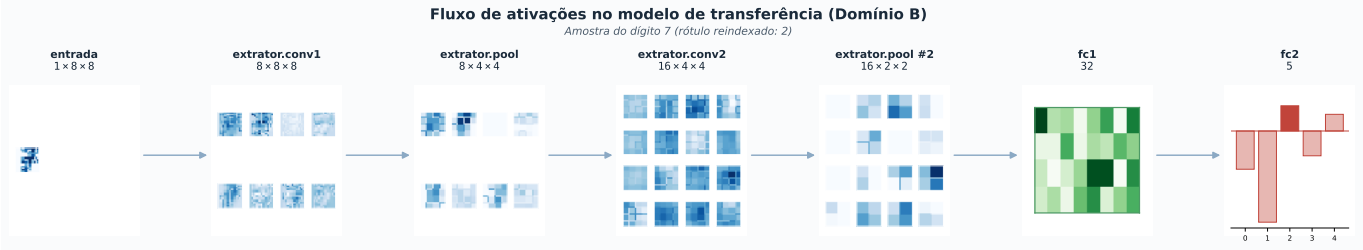


Figura 9.22: Flujo de activaciones y transformaciones dimensionales de los tensores en el modelo de transferencia de aprendizaje al procesar una muestra de prueba del Dominio B (dígito 7, reindexado a clase 2).

```
Capas capturadas en el modelo de transferencia:
 ['extrator.conv1', 'extrator.pool', 'extrator.conv2', 'extrator.pool #2', 'fc1', 'fc2']
```

Análisis del Experimento 1

El modelo entrenado desde cero alcanza una exactitud superior al modelo con transferencia de aprendizaje y extractor congelado. Este resultado caracteriza un caso de **transferencia negativa** (*negative transfer*) y se debe a tres factores:

1. **Baja capacidad:** El extractor posee únicamente 16 filtros 3×3, insuficientes para aprender representaciones generalizables.
2. **Especialización en el dominio:** El entrenamiento con los dígitos 0 a 4 produce filtros poco discriminativos para los dígitos 5 a 9.
3. **Ausencia de adaptación:** El congelamiento impide que el extractor ajuste sus filtros a la nueva tarea.

Provocación Pedagógica

Este experimento utiliza un extractor pequeño entrenado en un dominio restringido. ¿El resultado sería diferente si el extractor hubiera aprendido sus representaciones en una base con millones de imágenes y gran diversidad de objetos?

Experimento 2 — Cuando la Transferencia Realmente Funciona (*ResNet-18* Preentrenada)

El segundo experimento repite la misma estructura del primero — pocos ejemplos de entrenamiento, dos clases, comparación entre estrategias —, pero cambia el extractor artesanal de 16 filtros por la ***ResNet-18***, una arquitectura de 18 capas preentrenada en *ImageNet* ($1,4$ millones de imágenes, 1.000 categorías), y el *dataset* sintético de dígitos por fotografías reales del **Oxford-IIIT Pet Dataset** (PARKHI, 2012).

La tarea: distinguir dos razas caninas — **Pug** y **Boxer** — a partir de solo 15 fotografías de entrenamiento por clase.

¿Por qué este escenario?

El desafío aquí no es la similitud visual entre las razas — Pug y Boxer tienen portes y proporciones bien distintas —, sino la escasez de datos: apenas 30 fotografías reales en total, sin ninguna imagen

sintética. Es el tipo de problema de bajo presupuesto de datos que motiva, en la práctica, el uso de redes preentrenadas: no hay tiempo ni recursos para fotografiar y etiquetar miles de perros antes de entrenar un clasificador desde cero.

Bloco 1: Carga del *Dataset* Real y Muestreo Disperso

1. **Fuente:** el *Oxford-IIIT Pet Dataset* (PARKHI, 2012) se carga a través de `torchvision.datasets.OxfordIIITPe` que descarga automáticamente las 7.349 fotografías y sus etiquetas de raza en la primera ejecución.
2. **Filtrado:** solo se mantienen las dos razas de interés (`Pug`, `Boxer`).
3. **Escasez deliberada:** solo se sortean 15 fotografías de entrenamiento por clase (30 en total) — el resto compone el conjunto de prueba, utilizado exclusivamente para evaluación.

La Figura 9.23 muestra muestras de entrenamiento de cada raza.

```
import random

RACAS_ALVO = ["Pug", "Boxer"]
N_TREINO_POR_CLASSE = 15
N_TESTE_POR_CLASSE = 20

# 1. Descarga del dataset completo (37 razas) - licencia CC BY-SA 4.0
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. Filtrado de las dos razas de interés, separadas por clase
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. Muestreo: pocas imágenes de entrenamiento, más imágenes de prueba
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"Entrenamiento: {len(imgs_treino)} imágenes | Prueba: {len(imgs_teste)} imágenes")

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))
```

Figura 9.23

Bloco 2: Tres Estrategias sobre la Misma Arquitectura

Para aislar el efecto de la transferencia de aprendizaje, las tres estrategias reutilizan **exactamente la misma arquitectura** (*ResNet-18*), variando únicamente el origen de los pesos y qué parámetros permanecen entrenables:

1. **`do_zero`:** pesos aleatorios (`weights=None`) — equivalente a entrenar la arquitectura de la *ResNet-18* enteramente desde cero, como en el Bloque 2 del Experimento 1.
2. **`congelado`:** pesos preentrenados en *ImageNet*, con `requires_grad = False` en todas las capas convolucionales — solo la nueva capa final se entrena.
3. **`fine_tuning`:** pesos preentrenados en *ImageNet* como punto de partida, pero **sin** congelamiento — toda la red se ajusta al nuevo dominio, con una tasa de aprendizaje baja para no destruir el conocimiento previo.

En todos los casos, la capa final `fc` se reemplaza por `nn.Linear(fc.in_features, 2)`, correspondiente a las dos razas de destino.

```
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")
```

Bloque 3: Entrenamiento Comparativo y Análisis de Precisión

Reutilizando las funciones genéricas `treinar` y `calcular_acuracia`, definidas en el Bloque 2 del Experimento 1, los tres modelos se entrenan sobre el mismo conjunto de 30 fotografías y se evalúan en el conjunto de prueba (imágenes nunca vistas durante el entrenamiento):

- El modelo `do_zero` tiende a **sobreajustarse** rápidamente a las 30 fotografías de entrenamiento, sin generalizar al conjunto de prueba — 30 ejemplos son drásticamente insuficientes para ajustar los 11 millones de parámetros de la *ResNet-18* desde cero.
- El modelo `congelado` ya debería alcanzar una precisión considerablemente superior, ya que reutiliza, sin ningún ajuste, características visuales genéricas (bordes, texturas, contornos) aprendidas en *ImageNet* — solo la nueva capa lineal necesita ajustarse a las 30 fotografías.
- El modelo `fine_tuning` tiende a igualar o superar al extractor congelado, ya que parte del mismo conocimiento previo, pero aún permite un ajuste fino de toda la red a las particularidades visuales de las razas.

La Figura 9.24 resume los tres resultados.

```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}%")

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")
```

```python
plt.xticks(rotation=10)
plt.legend()
plt.tight_layout()
plt.show()
```

Figura 9.24

Bloque 4: Inspección Cualitativa de las Predicciones

Como en el Experimento 1 y en la sección siguiente sobre diagnóstico foliar, es instructivo observar individualmente algunas predicciones del mejor modelo (típicamente `fine_tuning` o `congelado`) sobre fotografías reales de prueba, comparando la etiqueta prevista con la raza real.

```
melhor_modelo = modelo_fine_tuning  # o modelo_congelado, según el resultado del Bloque 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)
        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))
```

Figura 9.25

```
# Limpieza explícita de los datos descargados y liberación de memoria
if FLAG_LIMPAR_DADOS:
    if os.path.exists('./dados_pets'):
        shutil.rmtree('./dados_pets')
        print('  Directorio de datos temporales ./dados_pets eliminado con éxito.')

    if torch.cuda.is_available():
        torch.cuda.empty_cache()
```

Experimento 3 — Diagnóstico Fitosanitario con Transferencia de Aprendizaje

El experimento anterior mostró que una *ResNet-18* preentrenada en **ImageNet** puede adaptarse a una nueva tarea utilizando pocas muestras. Ahora, la misma estrategia se aplica a un problema de diagnóstico fitosanitario. La *ResNet-18* debe clasificar imágenes de hojas en tres categorías: `["folha_saudavel", "folha_doente", "sintoma_desconhecido"]`.

- **`folha_saudavel`:** hoja sin lesiones visibles.
- **`folha_doente`:** hoja con manchas oscuras que simulan una enfermedad fúngica.
- **`sintoma_desconhecido`:** hoja con clorosis amarillenta, que representa un patrón diferente de la enfermedad conocida.

¿Por qué este escenario?

El diagnóstico fitosanitario constituye una importante aplicación de la VC en la agricultura de precisión. Un modelo preentrenado en **ImageNet** puede reutilizar características como bordes, texturas y patrones de color para aprender esta nueva tarea con pocas imágenes.

Bloque 1: Generación del *Dataset* Sintético

Este bloque genera un conjunto sintético con **30** imágenes por clase para entrenamiento y **8** para validación, totalizando **90** y **24** imágenes, respectivamente.

1. **Generación de la hoja:** La función `desenha_folha_base` crea el contorno de la hoja, variando tamaño, orientación y tonalidad de verde.
2. **Simulación de la enfermedad:** La función `aplica_manchas_doenca` añade manchas oscuras irregulares que simulan lesiones fúngicas.
3. **Simulación de otro síntoma:** La función `aplica_sintoma_desconhecido` añade regiones amarillentas que representan un patrón distinto de la enfermedad conocida.
4. **Visualización de las muestras:** La Figura 9.26 presenta ejemplos de las tres categorías del conjunto sintético.

```
CLASSES_FOLHA = ["folha_saudavel", "folha_doente", "sintoma_desconhecido"]


def desenha_folha_base(tam_img, rng):
    '''Dibuja el contorno oval de una hoja verde con nervadura central,
    con pequeñas variaciones de tono, tamaño y orientación entre muestras.'''
    img = np.full((tam_img, tam_img, 3), 245, dtype=np.uint8)  # fondo claro
    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)  # nervadura central
    return img, (cx, cy, eixo_a, eixo_b, angulo)


def aplica_manchas_doenca(img, centro_folha, rng, n_manchas=(4, 8)):
    '''Simula lesiones foliares: manchas oscuras de bordes irregulares
    (patrón típico de enfermedades fúngicas).'''
    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):
    '''Simula un patrón distintivo (moteado amarillento/clorosis), diferente
    de las manchas oscuras de la enfermedad conocida.'''
    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
```

```python
def aplica_ruido_sal_pimenta(img, prop_ruido=0.02, rng=None):
    '''Aplica ruido sal (puntos blancos) y pimienta (puntos negros) aleatorios.
    prop_ruido: fracción de píxeles alterados (ej: 0.02 = 2% de los píxeles).'''
    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

    # Aplica Sal (Blanco - [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]

    # Aplica Pimienta (Negro - [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)  # leve variación de exposición
    img = np.clip(img.astype(np.int16) + ruido_exp, 0, 255).astype(np.uint8)

    # Aplicación del ruido Sal y Pimienta
    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"Entrenamiento: {len(imgs_treino)} imágenes ({N_POR_CLASSE_TREINO} por clase) | "
      f"Validación: {len(imgs_val)} imágenes ({N_POR_CLASSE_VAL} por clase)")

# Visualización de 2 muestras de cada clase (6 imágenes en 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))
```

```
Entrenamiento: 90 imágenes (30 por clase) | Validación: 24 imágenes (8 por clase)
```

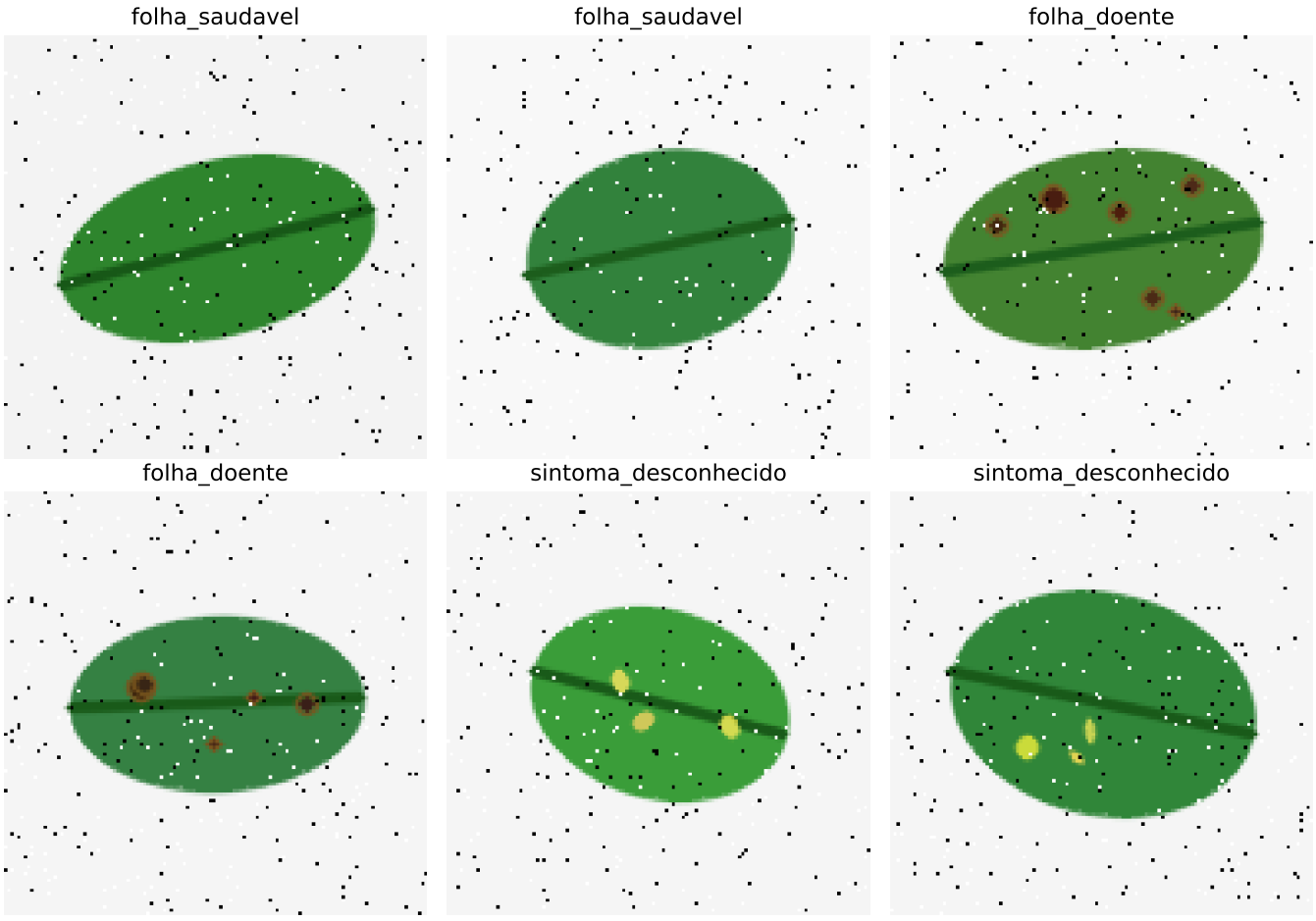


Figura 9.26: Muestras sintéticas del *dataset* de diagnóstico foliar: hoja sana, hoja enferma (manchas oscuras) y síntoma desconocido (clorosis amarillenta) con ruido sal y pimienta.

Trampa Común

La transferencia de aprendizaje exige que cada clase presente patrones visuales distintos. Repetir la misma imagen con etiquetas diferentes impide que la capa clasificadora aprenda una frontera de decisión, pues el extractor genera prácticamente las mismas características para todas las muestras.
En este experimento, cada imagen se genera de forma independiente, con patrones visuales compatibles con su clase (hoja sana, lesiones fúngicas o clorosis), proporcionando información suficiente para el entrenamiento de la capa clasificadora.

Bloque 2: Preparación de los Tensores y Adaptación de la Arquitectura

Modelos como la *ResNet-18* requieren imágenes en color de 224×224 *píxeles* normalizadas según las estadísticas de *ImageNet* ($\mu = [0,485; 0,456; 0,406]$ y $\sigma = [0,229; 0,224; 0,225]$).

1. **Transformación de Entrada (`transforms.Compose`):** se aplica el redimensionamiento y la normalización estándar a cada imagen del *dataset* sintético, produciendo los tensores `X_treino`/`X_val` y las etiquetas `y_treino`/`y_val` — cada ejemplo es una imagen genuinamente distinta, asociada a la etiqueta correcta de su clase.
2. **Congelamiento del Extractor:** el bucle `for p in modelo_resnet.parameters(): p.requires_grad = False` desactiva los gradientes en las capas convolucionales preentrenadas.
3. **Nueva Capa Final:** la capa `modelo_resnet.fc` se sustituye por una nueva instancia `nn.Linear(modelo_resnet.fc.in_features, n_classes_destino)`, recién inicializada y con gradientes activos por defecto. El número de características de entrada se obtiene dinámicamente a partir de la propia capa original (`in_features`, igual a 512 en la ResNet-18), en lugar de fijarse manualmente en el código — práctica recomendada, ya que hace que el fragmento sea reutilizable para otras variantes de la arquitectura sin modificaciones.

```
# 1. Pipeline de transformaciones esperadas por la 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. Conversión del dataset sintético (Bloque 1) en tensores normalizados
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. Carga de la ResNet-18 preentrenada y congelamiento del 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. Sustitución de la capa final para las 3 nuevas clases de destino
modelo_resnet.fc = nn.Linear(modelo_resnet.fc.in_features, n_classes_destino)

print(f"Nueva capa final: {modelo_resnet.fc}")
```

Bloco 3: Bucle de Entrenamiento y Evaluación

Con el extractor congelado y la nueva capa de salida debidamente acoplada, se ejecuta el ajuste fino (*fine-tuning*) de la nueva cabeza de clasificación.

1. **Optimización Enfocada:** el optimizador *Adam* recibe estrictamente `modelo_resnet.fc.parameters()`, actualizando únicamente la nueva capa de salida — el resto de la red permanece congelado, según lo definido en el Bloco 2.

2. **Ejecución del Bucle:** en cada época, el modelo itera sobre los lotes de entrenamiento, calcula la pérdida por Entropía Cruzada y ajusta los pesos de la capa final; a continuación, se evalúa la precisión en el conjunto de validación (imágenes nunca vistas durante el entrenamiento).
3. **Curvas de Entrenamiento:** la Figura 9.27 acompaña la evolución de la pérdida de entrenamiento y de la precisión de validación a lo largo de las épocas — como las tres clases son visualmente distintas entre sí, se espera una convergencia genuina, muy por encima del umbral de 33% correspondiente a una elección aleatoria entre 3 clases.

```
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"Época {epoca+1}/{epocas} - pérdida: {historico_perda[-1]:.4f} - ",
          f" precisión_val: {acc*100:.1f}%")

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",
)
```

Figura 9.27

Bloque 4: Inspección Cualitativa de las Predicciones

Además de la curva de precisión agregada, resulta instructivo observar individualmente algunas predicciones del modelo sobre el conjunto de validación, comparando la etiqueta prevista con la etiqueta real. La Figura 9.28 muestra dos muestras de cada clase.

```
modelo_resnet.eval()

# Dos muestras de cada clase en el conjunto de validación
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))
```

Figura 9.28

Análisis del Experimento 3

La *ResNet-18* alcanza una alta precisión incluso utilizando un conjunto reducido de imágenes sintéticas. Este resultado muestra que las representaciones aprendidas en **ImageNet** siguen siendo útiles en un dominio completamente diferente, requiriendo solo la adaptación de la capa clasificadora.

El experimento también ilustra una situación común en aplicaciones reales, en las cuales la disponibilidad de datos etiquetados es limitada. En estos escenarios, la transferencia de aprendizaje reduce el tiempo de entrenamiento y permite obtener modelos con buen rendimiento incluso sin entrenar toda la red.

Síntesis Comparativa — ¿Cuándo Funciona la Transferencia de Aprendizaje?

Los tres experimentos muestran que la transferencia de aprendizaje depende de la capacidad de generalización del extractor de características.

- **Experimento 1:** un extractor pequeño, entrenado en un dominio restringido, aprende representaciones poco generalizables y puede producir **transferencia negativa**.
- **Experimento 2:** una *ResNet-18* preentrenada en **ImageNet** transfiere representaciones generales a una tarea de clasificación de razas de perros y gatos, alcanzando una alta precisión con pocas muestras.
- **Experimento 3:** la misma estrategia adapta el modelo a un problema de diagnóstico fitosanitario, mostrando que un único extractor puede servir como base para diferentes dominios de aplicación.

Comparativo de los Enfoques de Transferencia

La Tabla 9.2 resume los resultados obtenidos en los tres experimentos.

Tabla 9.2: Comparación entre los tres escenarios de transferencia de aprendizaje presentados en esta sección.

Aspecto	Experimento 1	Experimento 2	Experimento 3
Extractor	CNN pequeña	*ResNet-18*	*ResNet-18*
Entrenamiento del extractor	Dígitos (0–4)	**ImageNet**	**ImageNet**
Capacidad de generalización	Baja	Alta	Alta
Nueva tarea	Dígitos (5–9)	Razas de perros y gatos	Diagnóstico fitosanitario
Resultado	Transferencia negativa	Transferencia positiva	Transferencia positiva

9.5.2 Detección de Objetos

Los experimentos anteriores mostraron cómo la transferencia de aprendizaje adapta modelos preentrenados para tareas de clasificación de imágenes. El mismo principio también fundamenta arquitecturas de detección de objetos, en las cuales un extractor de características preentrenado proporciona representaciones visuales generales, mientras que módulos especializados localizan y clasifican los objetos en la imagen.

Las próximas secciones presentan la **Faster R-CNN** como ejemplo de detector preentrenado utilizado directamente para inferencia y, a continuación, un experimento completo de ajuste fino con la arquitectura **YOLO**.

9.5.2.1 Faster R-CNN: Detector Preentrenado

La detección de objetos extiende el uso de modelos preentrenados a una tarea más compleja que la clasificación. La **Faster R-CNN** utiliza una CNN preentrenada, como la *ResNet-50*, como **extractor de características** (*backbone*) y añade módulos especializados para localizar y clasificar objetos.

Sobre este extractor, la arquitectura incorpora dos "cabezas" principales:

- ***Region Proposal Network* (RPN):** propone regiones de la imagen con alta probabilidad de contener objetos.
- **Cabeza clasificadora:** refina esas regiones, asigna una clase a cada objeto y ajusta sus cajas delimitadoras.

El siguiente código utiliza una Faster R-CNN con pesos preentrenados en **COCO** para detectar objetos en una imagen, produciendo sus clases, coordenadas y puntuaciones de confianza.

> ℹ 🔍 ¿Dónde está la transferencia de aprendizaje aquí?
>
> A diferencia de los experimentos anteriores, este ejemplo **no realiza ajuste fino** (*fine-tuning*). El modelo ejecuta únicamente la inferencia (`eval()`), reutilizando directamente los pesos del *backbone*, de la RPN y de la cabeza clasificadora entrenados en **COCO**.
> La adaptación a un nuevo dominio exigiría reemplazar la capa `box_predictor` por una nueva cabeza de clasificación, compatible con las clases de la aplicación, y entrenarla sobre un conjunto de imágenes anotadas. Este procedimiento sigue el mismo principio presentado en la sección de transferencia de aprendizaje y constituye el flujo habitual para aplicaciones específicas, como detección de plagas, defectos de fabricación o vehículos.

1. **Categorías de COCO:** El código recupera los nombres de las clases a partir de las meta-informaciones de los pesos (`FasterRCNN_ResNet50_FPN_Weights.DEFAULT.meta["categories"]`). Aunque esta lista contenga 91 entradas por razones históricas del formato de anotación de COCO, solo 80 corresponden a categorías de objetos.

2. **Inferencia:** La imagen cargada por `mm.read()` se convierte en *tensor* y se procesa mediante el modelo en modo de evaluación (`eval()`). El código mantiene únicamente las detecciones con confianza superior

a 80%.

3. **Anotación de la imagen:** Para cada objeto detectado, el código dibuja la caja delimitadora (`cv2.rectangle`) y escribe la clase predicha y su confianza (`cv2.putText`).
4. **Visualización:** La Figura 9.29 presenta la imagen anotada con las detecciones realizadas por el modelo.

9.5.2.2 Detección de Objetos y Transferencia de Aprendizaje con YOLO

Las secciones anteriores aplicaron la transferencia de aprendizaje a problemas de **clasificación de imágenes**, en los que el modelo asocia una única etiqueta a la imagen completa. En esta sección, el mismo principio se extiende a la **detección de objetos**, tarea que exige identificar simultáneamente **qué** está presente en la imagen y **dónde** se encuentra cada objeto.

La subsección anterior presentó la **Faster R-CNN** como ejemplo de detector preentrenado utilizado directamente para inferencia, sin ninguna adaptación al nuevo dominio. En este experimento, el modelo pasa por una etapa de **ajuste fino** (*fine-tuning*): se parte de una arquitectura **YOLO** (*You Only Look Once*) preentrenada en el conjunto **COCO** y se adapta la red para detectar y clasificar objetos de un nuevo dominio.

A diferencia de la Faster R-CNN, que realiza la detección en dos etapas, la familia **YOLO** adopta una arquitectura de etapa única (*single-stage detector*), estimando, en una única propagación por la red, las cajas delimitadoras (*bounding boxes*), la confianza de cada detección y la clase correspondiente. Esta estrategia reduce el costo computacional y hace viables las aplicaciones en tiempo real.

Como ejemplo, el experimento utiliza un conjunto sintético de formas geométricas (triángulos, cuadrados, estrellas, entre otras), con variaciones de color, tamaño, rotación y degradación por ruido de tipo sal y pimienta.

Bloco 1: Generación del *Dataset* Sintético

El siguiente bloque genera un conjunto sintético para el entrenamiento y la evaluación del detector. Cada imagen contiene entre uno y tres objetos pertenecientes a una de las nueve clases:

```
CLASSES = [
    'Triangle', 'Square', 'Pentagon', 'Hexagon',
    'Heptagon', 'Circle', 'Ellipse', 'Star', 'Cross'
]
```

1. **Generación de las formas:** Las funciones `poligono_regular`, `poligono_estrela` y `poligono_cruz` construyen las coordenadas de los objetos. La función `desenha_objeto` dibuja cada forma con posición, tamaño, orientación y color aleatorios y calcula su *bounding box*.
2. **Anotación en el formato YOLO:** La función `gera_imagem_ruidosa` genera entre uno y tres objetos por imagen y convierte cada *bounding box* al formato YOLO, representado por la clase y las coordenadas normalizadas del centro, ancho y alto.
3. **Degradación de la imagen:** La función `adiciona_ruido_sal_pimenta` añade ruido impulsivo, simulando imperfecciones de adquisición.
4. **Visualización de las muestras:** La Figura 9.30 presenta ejemplos del conjunto sintético con las *bounding boxes* superpuestas mediante la función `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):
```

```python
# 1. Carga de la imagen y de los nombres de las categorías del COCO
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"]  # Mapeo índice -> nombre de la clase

# 2. Carga del modelo Faster R-CNN preentrenado
modelo_detection = fasterrcnn_resnet50_fpn(weights=pesos_coco).eval()

# 3. Ejecución de la inferencia sin cálculo de gradientes
with torch.no_grad():
    predicao = modelo_detection([to_tensor(img)])[0]

# 4. Filtrado de las detecciones con confianza superior al 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. Dibujo de las cajas delimitadoras y de las etiquetas de clase
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}"

    # Dibuja el rectángulo rojo (RGB: 255, 0, 0) con un grosor de 3 píxeles
    cv2.rectangle(img_com_caixas, (x1, y1), (x2, y2), (255, 0, 0), 3)

    # Escribe la clase y la confianza encima de la caja delimitadora
    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. Visualización gráfica de la imagen resultante
mm.show(img_com_caixas, title="Faster R-CNN (COCO) - Detección con Clase y Confianza")
```

Figura 9.29

```
    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:  # Elipse
        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)

    # Sal (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]

    # Pimienta (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)

        # Patrón YOLO: (clase, x_centro, y_centro, ancho, alto) normalizados
        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
```

```
# Generación de 5 muestras para visualización inicial en la parte superior del proyecto
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)

    # Escritura temporal de la anotación para lectura nativa mediante 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}")

# Visualización del panel de 5 muestras
mm.show(
    imgs_demo,
    titles=titulos_demo,
    cols=n_amostras_iniciais,
    figsize=(14, 3)
)
```

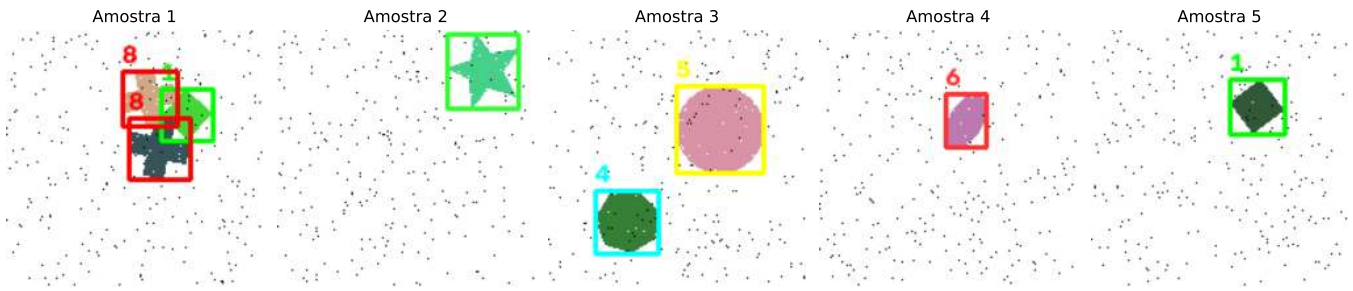


Figura 9.30: Muestras iniciales del *dataset* sintético ruidoso de objetos geométricos con las cajas delimitadoras del formato YOLO superpuestas.

Bloque 2: Organización del *Dataset* y Creación del Archivo `data.yaml`

Este bloque organiza el conjunto de datos en el formato esperado por la biblioteca **Ultralytics YOLO**. Las imágenes y las anotaciones se distribuyen en directorios separados para entrenamiento y validación, mientras que el archivo `data.yaml` reúne la información necesaria para el entrenamiento del detector.

```
shapes_dataset/
   data.yaml
   images/
       train/
       val/
   labels/
        train/
        val/
```

1. **Generación del conjunto de datos:** El código crea **90** imágenes para entrenamiento y **20** para validación. Para cada imagen, escribe un archivo `.txt` que contiene una línea por objeto, en el formato YOLO (`clase`, `x_c`, `y_c`, `ancho`, `alto`), con todas las coordenadas normalizadas.
2. **Organización de los archivos:** Las imágenes se almacenan en `images/train` e `images/val`, mientras que las anotaciones correspondientes se escriben en `labels/train` y `labels/val`, preservando el mismo nombre de archivo.
3. **Creación del archivo `data.yaml`:** El código genera automáticamente el archivo de configuración que contiene la ruta del *dataset*, los directorios de entrenamiento y validación, y el mapeo entre los índices numéricos y los nombres de las nueve clases.

```
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\nnames:\n"
```

```
    )
    for i, nome in enumerate(CLASSES):
        f.write(f"  {i}: {nome}\n")

print("Conjunto de datos ruidoso generado con éxito: 90 imágenes de entrenamiento y 20 de
validación.")
```

```
Conjunto de datos ruidoso generado con éxito: 90 imágenes de entrenamiento y 20 de validación.
```

Bloco 3: Preprocesamiento con Filtro de Mediana

Este bloque aplica un preprocesamiento para reducir el efecto del ruido de tipo sal y pimienta introducido en la generación del *dataset*. El **Filtro de Mediana** (`cv2.medianBlur`) elimina este tipo de degradación preservando mejor los bordes de los objetos que los filtros de suavizado convencionales.

1. **Filtrado de la imagen:** El código aplica un filtro de mediana con ventana 3×3 a la imagen ruidosa, reduciendo los *píxeles* impulsivos sin alterar las anotaciones del conjunto de datos.

2. **Visualización comparativa:** La Figura 9.31 compara la imagen original y la imagen filtrada, manteniendo las *bounding boxes* superpuestas mediante la función `mm.showBoundBox()`.

```
# 1. Carga de la primera muestra ruidosa del 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. Preprocesamiento con Filtro de Mediana (ventana 3x3)
img_filtrada = cv2.medianBlur(img_ruidosa, ksize=3)

# 3. Superposición de las bounding boxes con 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. Exhibición comparativa con mm.show
mm.show(
    [img_ruid_anotada, img_filt_anotada],
    titles=[
        "1. Imagen Ruidosa Original (Sal y Pimienta)",
        "2. Preprocesada (Filtro de Mediana 3x3)"
    ],
    cols=2,
    figsize=(9, 4)
)
```

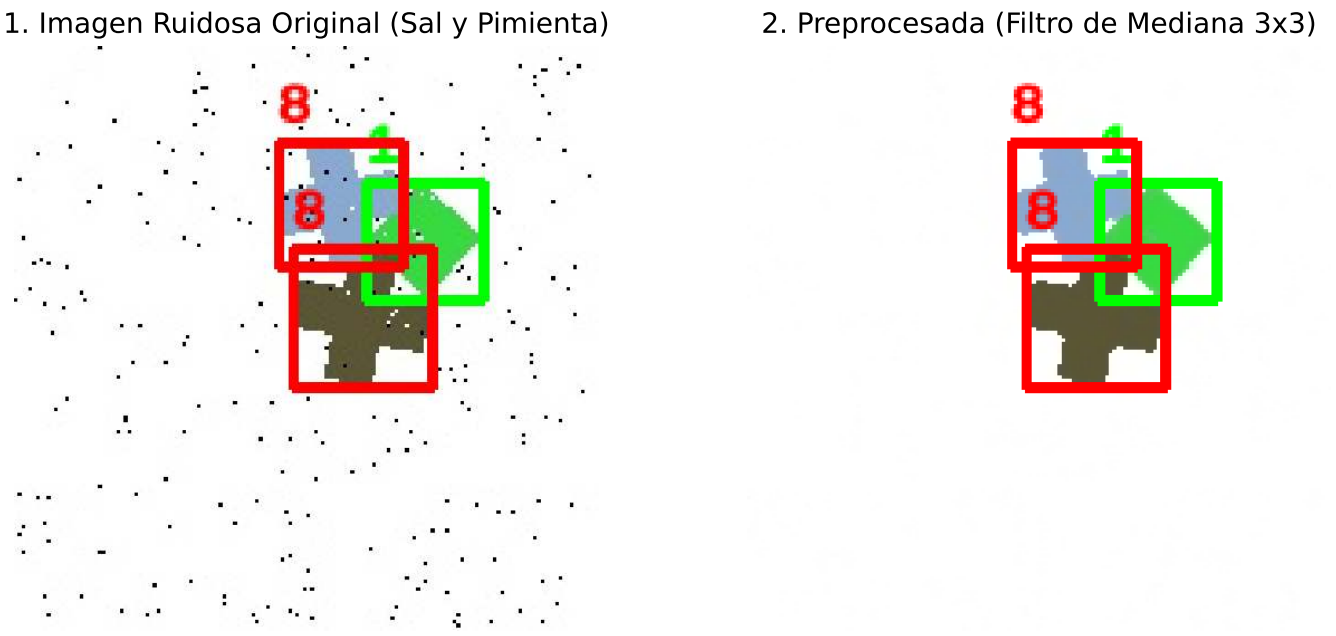


Figura 9.31: Comparación entre la imagen original con ruido tipo sal y pimienta y la imagen después de la aplicación del Filtro de Mediana (3x3). Las *bounding boxes* en el formato YOLO permanecen inalteradas.

Bloque 4: Preprocesamiento del *Dataset* y Ajuste Fino del YOLOv8

Este bloque aplica el Filtro de Mediana al conjunto de imágenes y realiza el ajuste fino (*fine-tuning*) del detector **YOLOv8n** preentrenado en el conjunto **COCO**. El filtrado reduce el efecto del ruido impulsivo introducido en la generación de las imágenes, mientras que el entrenamiento adapta los parámetros de la red al nuevo dominio de formas geométricas.

1. **Filtrado por lotes:** El código recorre los directorios `train` y `val` y aplica `cv2.medianBlur` con una ventana 3×3 en todas las imágenes, manteniendo las anotaciones YOLO originales.
2. **Ajuste fino del detector:** La red `YOLO("yolov8n.pt")`, inicialmente entrenada en COCO, se adapta al conjunto geométrico mediante la función `.train()`. El entrenamiento utiliza imágenes con resolución de 320×320 *píxeles* durante 30 épocas.
3. **Evaluación del modelo:** La función `.val()` calcula las métricas de detección en el conjunto de validación, incluyendo **precisión** (*precision*), **exhaustividad** (*recall*) y **mAP50** (*mean Average Precision* con umbral de IoU igual a $0,5$).

```
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

base_dir = "shapes_dataset"
base_dir_filt = "shapes_dataset_filtrado"

# 1. Crea una COPIA filtrada del conjunto de datos en una carpeta separada
#    (el conjunto de datos original en base_dir permanece ruidoso, intacto)
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)
            # guarda la versión filtrada en la carpeta NUEVA, no sobrescribe la original
            cv2.imwrite(f"{pasta_imgs_filt}/{nome_arq}", img_filtrada)

    # copia las etiquetas (no cambian con el filtro)
    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 apuntando al conjunto de datos FILTRADO
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("Preprocesamiento completado: conjunto de datos filtrado guardado en carpeta
separada.\n")

# Descargar modelo YOLOv8 preentrenado (yolov8n.pt) si aún no está presente
with contextlib.redirect_stdout(io.StringIO()), \
    contextlib.redirect_stderr(io.StringIO()):
    modelo_yolo = YOLO("yolov8n.pt")
print("Modelo YOLOv8 cargado.")
```

```
Preprocesamiento completado: conjunto de datos filtrado guardado en carpeta separada.

Modelo YOLOv8 cargado.
```

```
# 3. Callback personalizado para imprimir solo la época en ejecución
def on_train_epoch_start(trainer):
    epoch_atual = trainer.epoch + 1
    total_epochs = trainer.epochs
    # Escribe directamente en el stdout original (evadiendo el silenciador)
    sys.__stdout__.write(f"  Processando Época {epoch_atual}/{total_epochs}...\n")
    sys.__stdout__.flush()

# Añade el callback al modelo
modelo_yolo.add_callback("on_train_epoch_start", on_train_epoch_start)

# 4. Gestor de contexto para silenciar la basura de Ultralytics (C/C++ y 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

# Ejecución del Entrenamiento
print("--- Iniciando Entrenamiento YOLOv8 ---")
with silenciar_logs():
    resultados_treino = modelo_yolo.train(
        data=f"{base_dir_filt}/data.yaml",   # <-- entrena en el dataset filtrado
        epochs=30,
        imgsz=320,
        batch=16,
        device=device,
        verbose=False,
        plots=False
    )
    metricas = modelo_yolo.val(verbose=False)

# 5. Métricas Finales
precision = metricas.results_dict["metrics/precision(B)"]
recall = metricas.results_dict["metrics/recall(B)"]
map50 = metricas.results_dict["metrics/mAP50(B)"]

print("\n--- Rendimiento Optimizado del Modelo YOLOv8 ---")
print(f"Precisión: {precision*100:.2f}%")
print(f"Revocación (Recall): {recall*100:.2f}%")
print(f"mAP al 50% (IoU 0.50): {map50*100:.2f}%")
```

```
--- Iniciando Entrenamiento YOLOv8 ---

--- Rendimiento Optimizado del Modelo YOLOv8 ---
Precisión: 78.45%
Revocación (Recall): 90.19%
mAP al 50% (IoU 0.50): 95.63%
```

Bloque 5: Comparativo de Inferencia: Imagen Ruidosa e Imagen Restaurada

Tras el ajuste fino de YOLOv8 en el conjunto restaurado, se realiza una comparación visual entre la detección aplicada directamente sobre una imagen degradada por ruido sal y pimienta y la misma imagen después del Filtro de Mediana.

El objetivo es observar cómo una etapa simple de preprocesamiento puede influir en la calidad de las predicciones de un detector ya adaptado al nuevo dominio.

1. **Inferencia con la YOLO:** La función `modelo_yolo.predict()` ejecuta la detección en las dos versiones de la imagen, utilizando un umbral de confianza de 25% (`conf=0.25`).
2. **Visualización de las Predicciones:** La función `plot()` genera las imágenes anotadas con las cajas delimitadoras y las etiquetas previstas por el modelo. La Figura 9.32 presenta la comparación entre los dos escenarios.

```
# Limpieza explícita de los datos descargados y liberación de memoria
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))
```

```python
# 1. Carga de una muestra de prueba original (sin el filtro guardado en lote)
caminho_teste = f"{base_dir}/images/val/0002.jpg"
img_ruidosa_teste = mm.read(caminho_teste)

# 2. Aplicación puntual del Filtro de Mediana (3x3) para comparación
img_filtrada_teste = cv2.medianBlur(img_ruidosa_teste, ksize=3)


# 3. Inferencia con el modelo YOLOv8 entrenado
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. Extracción de las matrices anotadas por el generador del YOLO (conversión BGR -> RGB)

img_pred_ruid = cv2.cvtColor(pred_ruidosa.plot(), cv2.COLOR_BGR2RGB)
img_pred_filt = cv2.cvtColor(pred_filtrada.plot(), cv2.COLOR_BGR2RGB)

# 5. Exhibición comparativa estandarizada mediante mm.show
mm.show(
    [img_pred_ruid, img_pred_filt],

    titles=[
        f"Inferencia en la Imagen Ruidosa ({len(pred_ruidosa.boxes)} objetos)",
        f"Inferencia en la Imagen Filtrada ({len(pred_filtrada.boxes)} objetos)"
    ],
    cols=2,
    figsize=(10, 4)

)
```

Figura 9.32

```
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()
```

Un dominio mucho más distante que el de los dígitos

En el experimento de transferencia de aprendizaje entre dígitos manuscritos (dominios A y B), la tarea de origen y la de destino compartían estadísticas visuales muy cercanas: ambas eran trazos en tonos de gris sobre fondo uniforme. Aquí, la distancia entre dominios es mucho mayor — el YOLO fue preentrenado en **fotografías naturales coloridas** de COCO (personas, animales, vehículos, objetos cotidianos), y la tarea de destino consiste en **formas geométricas sintéticas, de color sólido y contorno bien definido**, sin textura, iluminación ni fondo complejo.
Aun así, la transferencia de aprendizaje es ventajosa: las capas iniciales de un detector entrenado en COCO aprenden filtros genéricos — detectores de bordes, esquinas y regiones de contraste — que siguen siendo útiles para delimitar el contorno de un triángulo o de una estrella, incluso si el contenido visual final es bastante distinto. Es por ello que permitir el ajuste fino (*fine-tuning*) de todas las capas, combinado con un preprocesamiento consistente entre entrenamiento e inferencia para atenuar el ruido sal y pimienta, posibilita tasas de precisión elevadas en la detección de objetos del nuevo dominio.

Usando un Conjunto de Datos Real

El *pipeline* anterior fue construido íntegramente en torno al **formato de anotación YOLO** (clase, x_{centro}, y_{centro}, ancho, alto, normalizados por el ancho y alto de la imagen) exactamente para que pueda reutilizarse sin modificaciones si el lector tiene acceso a un conjunto de imágenes reales anotadas de la misma manera — por ejemplo, un conjunto de imágenes de objetos geométricos fotografiados o renderizados, cada uno con su archivo `.txt` correspondiente en el mismo formato usado aquí. Para ello, bastaría con:

1. Organizar las imágenes reales en `shapes_dataset/images/train` y `shapes_dataset/images/val`, y los archivos `.txt` de anotación correspondientes en las carpetas `labels/train` y `labels/val` (un archivo de anotación por imagen, mismo nombre base, extensión `.txt`);
2. Ajustar el archivo `data.yaml` si el número o los nombres de las clases son diferentes;
3. Ejecutar las mismas celdas de entrenamiento, ajuste fino y visualización ya presentadas, sin ninguna otra modificación de código.

Esta separación entre **generación/organización de los datos** y **entrenamiento del modelo** es, en la práctica, la razón por la cual los formatos de anotación estandarizados (como el de YOLO) son tan ampliamente adoptados: permiten intercambiar el conjunto de datos de entrada — sintético por real, un dominio por otro — manteniendo inalterado todo el resto del *pipeline* de transferencia de aprendizaje.

9.5.3 Segmentación de Objetos

El mismo principio de transferencia de aprendizaje también fundamenta arquitecturas de segmentación de imágenes, en las cuales un extractor de características preentrenado proporciona representaciones visuales generales, mientras que una cabeza especializada realiza la clasificación densa, píxel a píxel.

Las siguientes secciones presentan la **DeepLabV3** como ejemplo de segmentador preentrenado utilizado directamente para inferencia y, a continuación, la arquitectura **U-Net**, entrenada desde cero y comparada con una línea base morfológica clásica.

9.5.3.1 DeepLabV3: Segmentador Preentrenado

En la segmentación semántica, el objetivo no se limita a la localización de los objetos mediante cajas delimitadoras (*bounding boxes*). La red asigna una clase a cada *píxel* de la imagen, produciendo un mapa de etiquetas con la misma resolución que la entrada. Arquitecturas como la **DeepLabV3**, con *backbone* **ResNet-50**, utilizan un extractor de características preentrenado y una cabeza especializada para realizar esta clasificación densa.

1. **Carga e inferencia:** El modelo `deeplabv3_resnet50(weights="DEFAULT")` carga pesos preentrenados en el conjunto **Pascal VOC**, que define 21 clases de segmentación. El código convierte la imagen en *tensor*, añade la dimensión de *batch* (`unsqueeze(0)`) y ejecuta la inferencia.
2. **Mapa de clases:** La salida del modelo posee dimensión $(1, 21, H, W)$, conteniendo un valor para cada clase en cada *píxel*. La operación `.argmax(dim=1)` selecciona la clase de mayor respuesta en cada posición, generando una matriz bidimensional de etiquetas con dimensiones (H, W).
3. **Visualización:** El código convierte el mapa de etiquetas al formato esperado por `mm.show()`, que muestra el resultado de la segmentación en la Figura 9.33.

```
# 1. Carga de la imagen (retorna 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. Carga del modelo DeepLabV3 preentrenado en modo de evaluación
modelo_segmentacao = deeplabv3_resnet50(weights="DEFAULT").eval()

# 3. Ejecución de la inferencia sin cálculo de gradientes
with torch.no_grad():
    tensor_entrada = to_tensor(img).unsqueeze(0)  # Formato (1, C, H, W)
    saida = modelo_segmentacao(tensor_entrada)["out"]

    # Selección de la clase con mayor probabilidad por píxel (argmax en el eje de los canales)
    mapa_classes = saida.argmax(dim=1).squeeze(0).byte().cpu().numpy()

# 4. Visualización del mapa de segmentación semántica
mm.show(
    [img,mapa_classes],
    title=["Imagen original","Segmentación Semántica (DeepLabV3)"]
)
```

Figura 9.33

9.5.3.2 Segmentación Semántica con Arquitectura U-Net

La subsección anterior presentó un modelo preentrenado (*DeepLabV3*) que produce directamente una etiqueta de clase por *píxel*. Esta sección completa la secuencia de proyectos prácticos — clasificación y detección — abordando la segmentación semántica implementada y entrenada desde cero con la **U-Net**, la arquitectura de referencia introducida por Ronneberger (2015).

En la clasificación, el mapa de características final se aplanaba (*flatten*) en un vector, descartando la información espacial en favor de una única etiqueta por imagen. La U-Net, por su parte, produce una salida con la **misma resolución espacial que la entrada**: un mapa bidimensional en el cual cada *píxel* recibe su propia clasificación. Este requisito — preservar el detalle espacial de alta resolución al mismo tiempo que se construye contexto semántico en capas profundas — motiva la arquitectura de codificador-decodificador con conexiones de atajo (*skip connections*).

El escenario utilizado simula la segmentación de nódulos en exámenes médicos sintéticos: imágenes en escala de grises contienen una región circular ("nódulo") superpuesta sobre un fondo, ambas contaminadas por

ruido gaussiano y con medias de intensidad muy cercanas — un desafío intencional de bajo contraste, ideal para demostrar la ganancia del aprendizaje espacial frente a la umbralización puntual.

Bloco 1: Gerador del Conjunto Sintético de Nódulos y Visualización Inicial

El siguiente generador produce pares (imagen, máscara): la imagen contiene una región circular de intensidad ligeramente superior a la del fondo, ambas afectadas por la misma desviación estándar de ruido gaussiano. La máscara binaria delimita exactamente la región del nódulo y sirve como verdad de referencia (*ground truth*).

1. **Construcción del Nódulo:** La función `gera_imagem_com_nodulo` superpone el nódulo al fondo sobre una matriz 64×64 y aplica ruido gaussiano.
2. **Visualización con `mm.show`:** La Figura 9.34 ilustra las dos primeras muestras y sus respectivas máscaras.

```
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


# Generación de los conjuntos de entrenamiento y validación
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"Conjunto de entrenamiento: {N_TREINO} imágenes | Conjunto de validación: {N_VAL}
    imágenes\n"
)

# Exhibición de muestras iniciales
mm.show(
```

```
    [imgs_treino[0], masks_treino[0], imgs_treino[1], masks_treino[1]],
    titles=["Imagen 1", "Máscara 1", "Imagen 2", "Máscara 2"],
    cols=4,
    figsize=(11, 3),
)
```

```
Conjunto de entrenamiento: 160 imágenes | Conjunto de validación: 40 imágenes
```

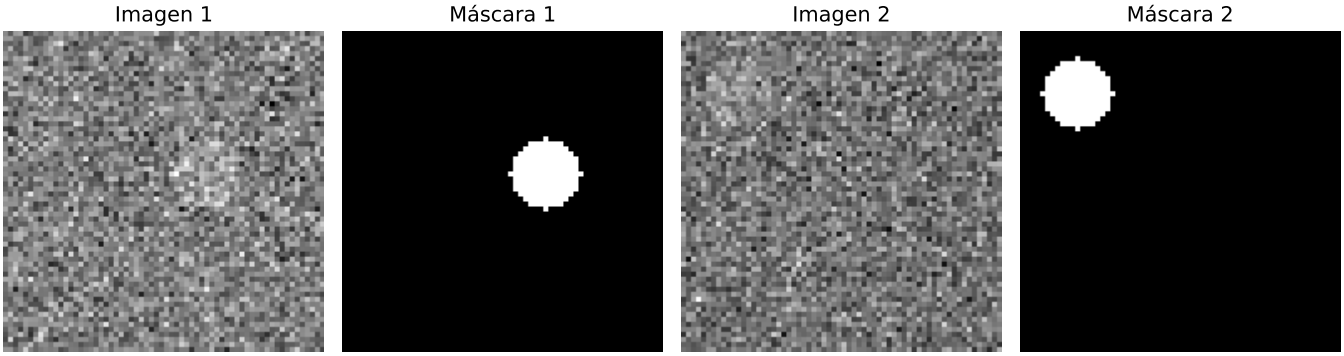


Figura 9.34: Muestras del conjunto de datos sintético de nódulos: imagen en escala de grises bajo ruido y su respectiva máscara binaria de referencia mostradas.

Bloco 2: Línea Base Clásica (Filtrado, Otsu y Morfología)

Antes de emplear la U-Net, se evalúa el desempeño de un *pipeline* morfológico clásico construido con la biblioteca `morph`: un **suavizador gaussiano** (`mm.blur`), una **umbralización de Otsu** (`mm.threshold`) y una **apertura morfológica** (`mm.open`) para la eliminación de ruidos aislados.

1. **Métrica de Intersección sobre Unión (IoU):** La función `iou_mascaras` calcula el grado de superposición *píxel a píxel* entre la predicción y la máscara real.
2. **Ejecución y Comparativa:** La Figura 9.35 muestra el resultado de la segmentación clásica en una imagen de prueba, demostrando las limitaciones del umbral global bajo bajo contraste.

```
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"IoU medio (línea de base clásica) en la validación: {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=["Imagen", "Máscara de Referencia", "Predicción Clásica"],
    cols=3,
    figsize=(9, 3.2),
)
```

```
IoU medio (línea de base clásica) en la validación: 0.7516
```

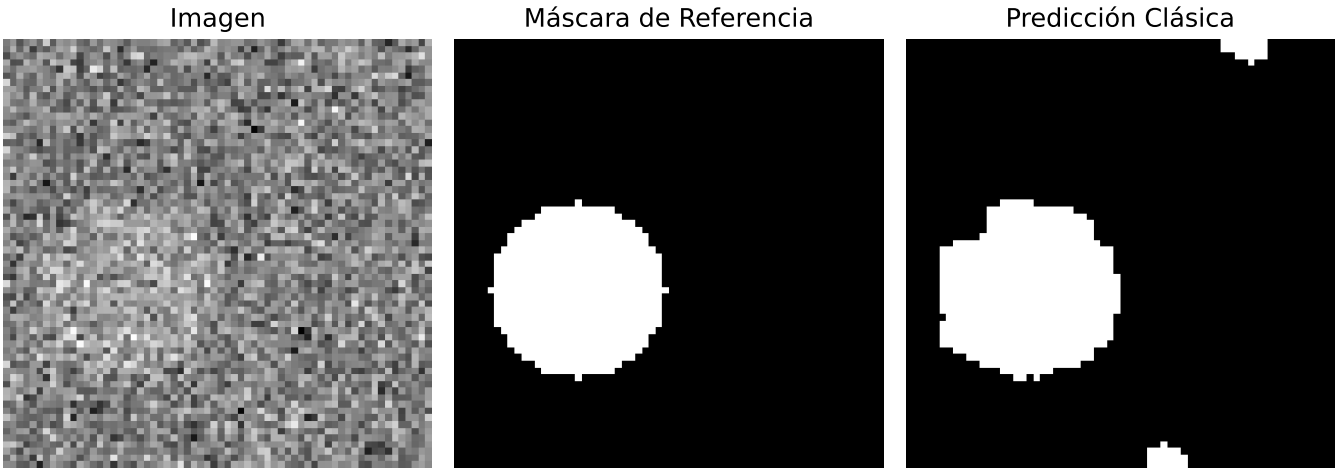


Figura 9.35: Línea de base clásica de segmentación: suavizado, umbralización de Otsu y apertura morfológica mostradas.

Bloque 3: Construcción de la Arquitectura U-Net y Funciones de Pérdida

La U-Net es una arquitectura en forma de "U" (de ahí su nombre), pensada específicamente para la segmentación de imágenes. Está formada por dos caminos que trabajan en conjunto:

- ↓ **Codificador (encoder):** desciende por la imagen, reduciendo la resolución espacial en cada etapa mientras extrae características cada vez más abstractas (bordes → texturas → formas → contexto).
- **Decodificador (decoder):** asciende de nuevo, reconstruyendo la resolución original mediante convoluciones transpuestas (*upsampling*), hasta generar una máscara del mismo tamaño que la imagen de entrada.

El elemento que hace especial a la U-Net son las **conexiones de atajo** (*skip connections*): estas llevan los mapas de características del codificador directamente a la etapa correspondiente del decodificador, en la misma resolución. Esto evita que los detalles finos —como contornos y bordes— se pierdan durante la compresión espacial.

Flujo general de la arquitectura:

```
Entrada


Codificador (Conv → Conv → Pool) × 3

      conexiones de atajo reinyectan mapas de alta resolución

Base (bottleneck)


Decodificador (Upsample → Concat → Conv → Conv) × 3
```

```
Salida 1×1 (logits)
```

Componentes principales:

1. **Bloque Convolucional Base (`BlocoConv`)** La unidad fundamental repetida en toda la red. Aplica dos convoluciones 3×3 en secuencia, cada una seguida de activación ReLU, con *padding* que preserva las dimensiones espaciales de la entrada. Este bloque aparece tanto en el codificador como en el decodificador.

2. **Convolución Transpuesta (`nn.ConvTranspose2d`)** Es la operación responsable del *upsampling* en el decodificador: en lugar de reducir la resolución espacial (como hace el `MaxPool2d` en el codificador), la aumenta, aprendiendo los pesos necesarios para "deshacer" la compresión y recuperar gradualmente el tamaño original de la imagen.

3. **Pérdida Combinada (BCE + Dice)** La función `perda_segmentacao` suma dos métricas complementarias:

   - **Entropía Cruzada Binaria (BCE):** evalúa el acierto *pixel a pixel*.
   - **Coeficiente de Dice:** evalúa la *superposición global* entre la máscara prevista y la real.

   Juntas, equilibran la precisión local con la fidelidad de la forma segmentada en su conjunto.

```python
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)
```

Bloque 4: Entrenamiento de la U-Net y Evaluación de Rendimiento

El entrenamiento se ejecuta durante 35 épocas utilizando el optimizador *Adam*. En cada época, se monitorea la Pérdida de Entrenamiento y el índice IoU promedio en el conjunto de validación.

1. **Bucle de Entrenamiento:** Se actualizan los parámetros con *mini-lotes* de 16 muestras.
2. **Evolución Gráfica:** La Figura 9.36 muestra el gráfico con el progreso de la pérdida y del IoU.

```
modelo_unet = UNetCompacta()
print(
    f"Parámetros entrenables de la U-Net: {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
```

```python
epocas_unet = 35
historico_perda_unet, historico_iou_unet = [], []

for epoca in range(epocas_unet):
    if epoca % 5 == 0:  # Imprime cada 5 épocas
        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"IoU medio final de la U-Net en la validación: {iou_unet_final:.4f}")

# Gráfico de la evolución del entrenamiento
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")
```

```
Parámetros entrenables de la U-Net: 120681
1 / 35
6 / 35
11 / 35
16 / 35
21 / 35
26 / 35
31 / 35
IoU medio final de la U-Net en la validación: 0.8876
```

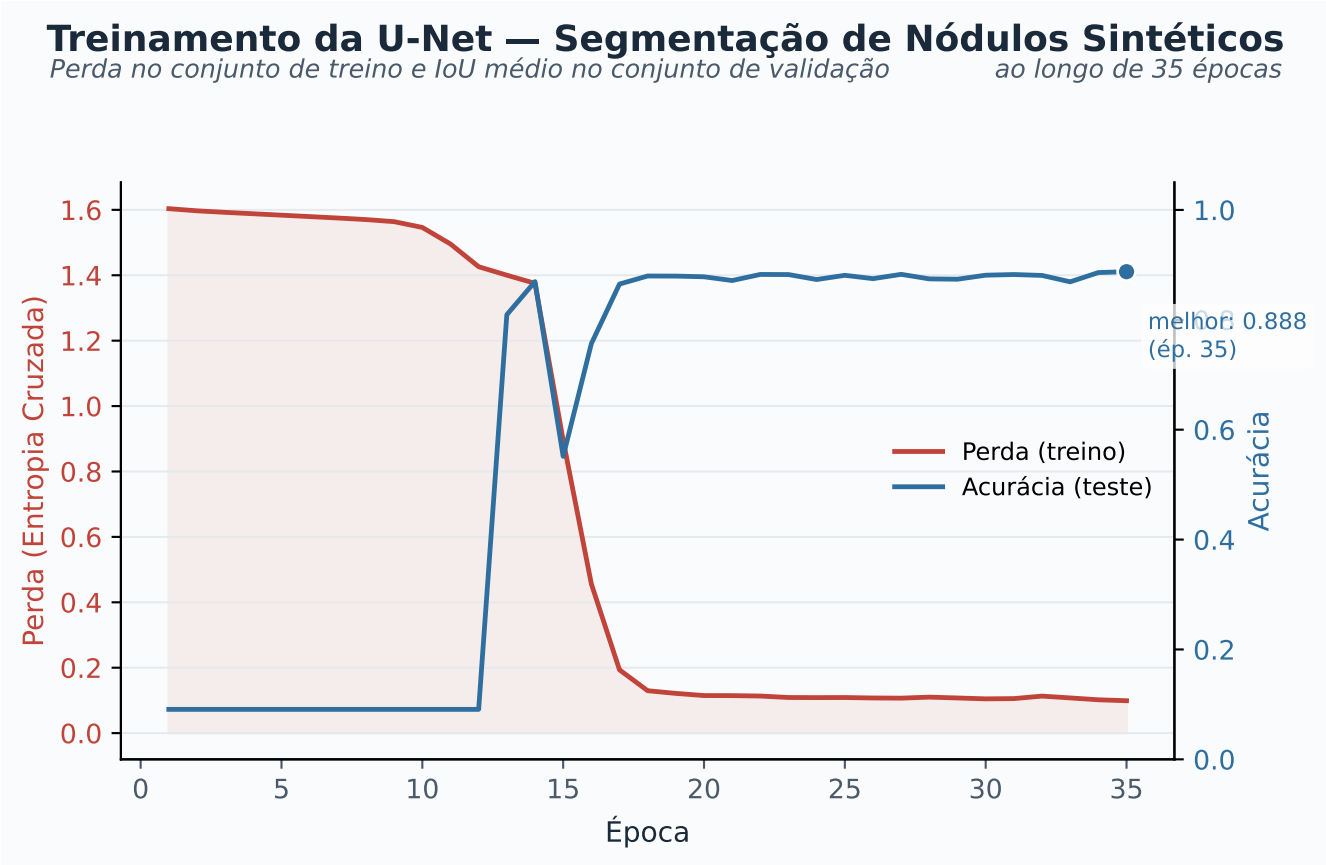


Figura 9.36: Evolución del entrenamiento de la U-Net compacta: reducción de la pérdida y elevación del índice IoU medio en el conjunto de validación a lo largo de 35 épocas.

Bloque 5: Comparativo Cuantitativo y Cualitativo (Clásico vs. U-Net)

La comparación entre el enfoque clásico y la U-Net evidencia la superioridad del aprendizaje de representaciones en escenarios de bajo contraste.

1. **Panel de Métricas:** El gráfico de barras en la Figura 9.37 contrasta el IoU medio de ambos métodos en la validación.
2. **Visualización Cualitativa:** La comparación visual en tres muestras demuestra cómo las conexiones de atajo recuperan el contorno del nódulo incluso bajo ruido acentuado.

```
# 1.  comparativo
print(f"IoU promedio (Suavizado clásico + Otsu): {iou_classico_medio:.4f}")
print(f"IoU promedio (U-Net): {iou_unet_final:.4f}")

# 2. Visualización cualitativa lado a lado en 3 muestras mediante 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),
)
```

```
IoU promedio (Suavizado clásico + Otsu): 0.7516
IoU promedio (U-Net): 0.8876
```

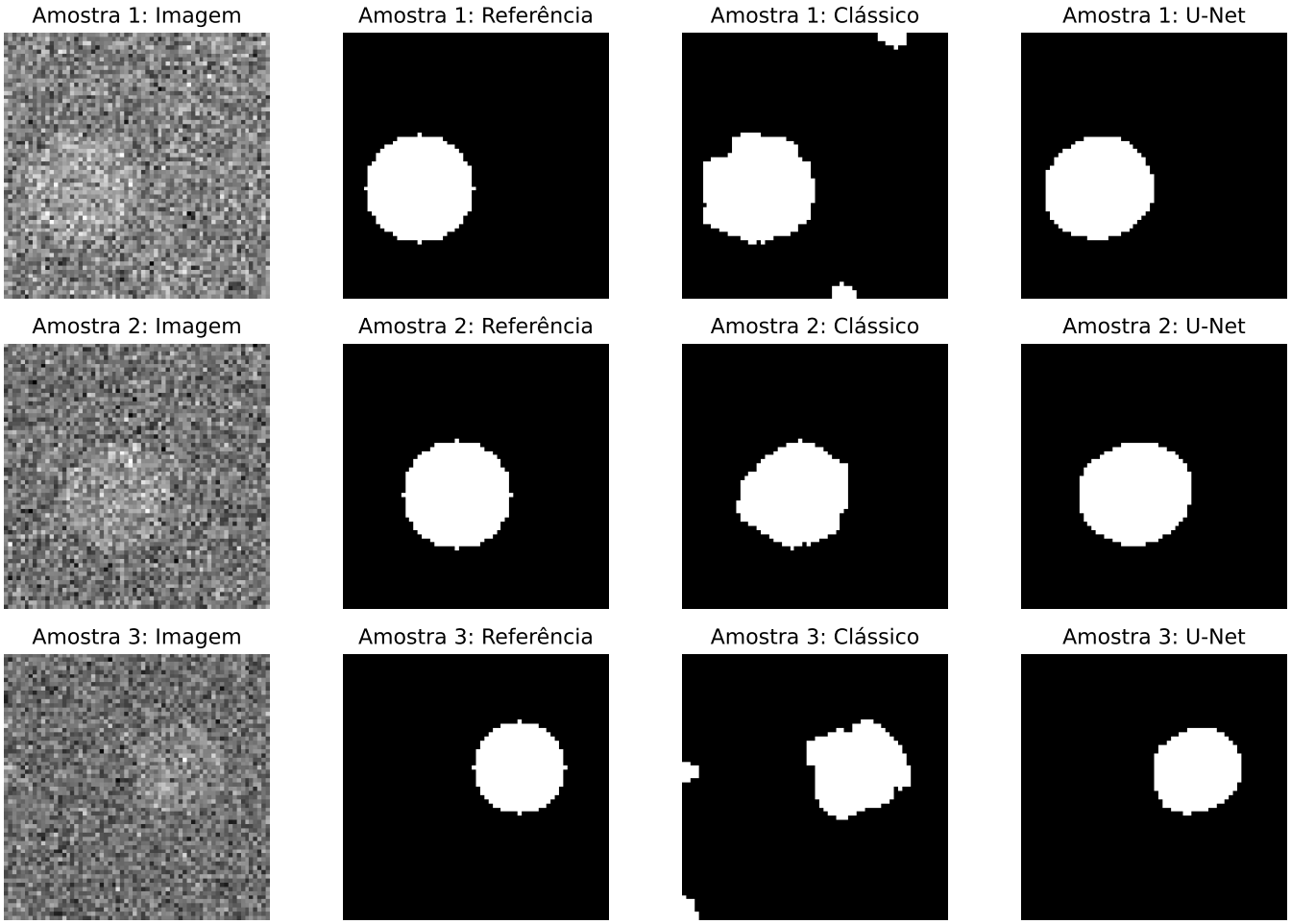


Figura 9.37: Comparativo cuantitativo (IoU promedio) y cualitativo entre el enfoque clásico y la U-Net entrenada en tres muestras de validación.

¿Por qué la U-Net supera a la umbralización fija?

La umbralización de Otsu aplica un valor de corte global sobre la intensidad local. Cuando la diferencia de media entre el nódulo y el fondo es pequeña frente al ruido gaussiano, esta regla comete errores sistemáticos en los bordes.
La U-Net supera esta limitación al combinar el contexto semántico amplio extraído por el codificador con los detalles espaciales finos preservados por las conexiones de atajo. Esto permite identificar la presencia del nódulo y delimitar sus contornos con precisión, incluso bajo ruido intenso.

9.5.4 Ingeniería de Datos para VC (*Roboflow*)

Una U-Net puede entrenarse a partir de imágenes anotadas en plataformas de ingeniería de datos para Visión por Computador (VC), como *Roboflow*. Estas plataformas permiten organizar conjuntos de datos, realizar anotaciones, aplicar etapas de preprocesamiento y *data augmentation*, entrenar modelos y exportar los datos en diferentes formatos. En esta sección, sin embargo, el ejemplo retoma la **detección de objetos geométricos**, utilizando conjuntos de datos ya presentados en este libro.

Dependencia de conexión y clave de API

Las celdas de esta sección requieren conexión a Internet y una clave de API gratuita de *Roboflow* (`app.roboflow.com`). En el *Workspace* del proyecto, acceda a **§ → Roboflow API** y copie la **Private API Key**.
Cree, en esta carpeta, el archivo `chave_roboflow.txt` que contenga **únicamente la clave**, sin comillas. Un modelo de este archivo está disponible en `chave_roboflow.txt.exemplo`.
Agregue `chave_roboflow.txt` al archivo `.gitignore`, ya que contiene una credencial de acceso que no debe versionarse ni compartirse.

9.5.4.1 Detección de objetos utilizando *Roboflow*

Roboflow permite realizar inferencia sobre imágenes locales, lo que posibilita evaluar el rendimiento del modelo alojado en la identificación de los objetos de interés.

Además de la inferencia, el *dataset* puede exportarse en el formato `png-mask-semantic`, en el cual cada imagen se acompaña de una máscara de segmentación semántica. En esta máscara, cada píxel representa la clase a la que pertenece el objeto correspondiente. Los pares imagen-máscara se utilizan como datos de entrenamiento para la `UNetCompacta`.

9.5.4.2 Conexión, *descarga* y verificación del *dataset*

El código establece conexión con el *Workspace* `mctest` y el proyecto `geometric-test00`, versión 6, y realiza la *descarga* del *dataset* hacia `dados/datasetRoboFlow`. A continuación, verifica la *forma* de las imágenes en cada *split*. La función de verificación se reutiliza posteriormente en la sección.

```
from roboflow import Roboflow
from pathlib import Path
from PIL import Image
from collections import Counter

def contar_shapes(raiz, splits=("train", "valid", "test")):
    """Cuenta el shape (alto, ancho, canales) de las imágenes por split."""
    for split in splits:
        pasta = raiz / split / "images"
        if not pasta.exists():
            print(f"{split}: carpeta no encontrada")
            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("Clave de Roboflow no encontrada: clave_roboflow.txt")
else:
    with open(chave) as f:
        api_key = f.read().strip()

    # Proyecto:
    # 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} | Nombre: {versao.name} | "
        f"Imágenes: {versao.images}"
    )

    raiz = Path("dados/datasetRoboFlow")
    versao.download("yolov8", location=str(raiz))

    print("Dataset:", raiz)
    contar_shapes(raiz)
```

9.5.4.3 Descargando el *dataset* de forma reproducible (alternativa)

Como alternativa al *dataset* obtenido mediante *Roboflow*, se puede utilizar un *dataset* disponible en un repositorio de GitHub, también organizado en los mismos *splits* y que contiene las mismas clases de objetos. Los conjuntos de datos, sin embargo, no son idénticos: las imágenes de GitHub tienen una resolución de `608×608` píxeles, mientras que las imágenes exportadas por *Roboflow* tienen `640×640` píxeles.

El código siguiente realiza la descarga del *dataset* desde GitHub, en caso de que aún no esté disponible localmente, y reutiliza `contar_shapes` para verificar las dimensiones de las imágenes en cada *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 Comparando una imagen de cada *dataset*

Los dos *datasets* poseen imágenes con resoluciones diferentes: `608×608` en GitHub y `640×640` en *Roboflow*. Para una comparación visual directa, las imágenes se redimensionan a la misma dimensión antes de ser mostradas lado a lado (Figura 9.38).

```
import cv2

if not chave.exists():
    print("Clave de Roboflow no encontrada: chave_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))

    # Redimensiona ambas para el mismo tamaño (el menor entre las dos)
    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}"],
    )
```

Figura 9.38

9.5.4.5 Inferencia con el modelo entrenado

El modelo entrenado en la versión 6 se utiliza para realizar la inferencia sobre una imagen de prueba local. La predicción considera los umbrales de confianza y de superposición empleados por la supresión de no máximos (*Non-Maximum Suppression*, NMS), y el resultado se presenta en la imagen anotada de la Figura 9.39.

9.5.4.6 Evaluando las predicciones con IoU y clase

Roboflow devuelve cada caja con las coordenadas del centro (`x`, `y`) en *píxeles* y la clase predicha, mientras que las etiquetas locales (`datos/conjunto_de_datos/prueba/etiquetas/00001.txt`) siguen el formato YOLO, con centro y dimensiones normalizadas en el intervalo `[0, 1]`. Antes de la comparación mediante `mm.IoU`, las cajas deben convertirse al mismo formato, con las coordenadas de la esquina superior izquierda y las dimensiones expresadas en *píxeles*.

Una predicción solo se considera correcta cuando la clase predicha coincide con la clase de la caja real y su *Intersection over Union* (IoU) es mayor o igual al umbral definido, adoptándose, en este ejemplo, `0,5` (50%) como valor predeterminado.

```python
# version.model está obsoleto; usar version.models()

if not chave.exists():
    print("Clave de Roboflow no encontrada: 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()  # incluye los cuadros detectados en resp["predictions"]

    altura, largura, _ = mm.read(img_path).shape
    print("Dimensiones de la imagen de prueba:", (altura, largura))

    pred.save("resultado.jpg")  # imagen anotada


    mm.show(
        mm.read("resultado.jpg"),
        title="Resultado de la inferencia con el modelo de Roboflow",
        figsize=(6, 6)
    )
```

Figura 9.39

! El `class_id` de *Roboflow* no corresponde al índice de las *etiquetas* locales

En la exportación, *Roboflow* reordena las clases en **orden alfabético** en el `data.yaml`, independientemente del orden utilizado en el proyecto original, mantenido en el `data.yaml` del GitHub. Así, `class_id = 0` corresponde a `Circulo` en la respuesta de la API, mientras que el mismo índice corresponde a `Triangulo` en las *etiquetas* locales.
Por ello, la comparación debe realizarse por el **nombre de la clase** (`p["class"]`), convirtiéndolo posteriormente al índice correspondiente en la lista local. Los `class_id` no deben compararse directamente.

```python
# resp, ancho y altura se definieron en la celda anterior

# Orden de clases usado en las etiquetas locales (datos/dataset/*/labels/*.txt)
CLASSES_LOCAIS = ["Triangulo", "Quadrado", "Pentagono", "Hexagono",
                  "Heptagono", "Circulo", "Elipse"]

def predicao_correta(pred: tuple, real: tuple, limiar: float = 0.5) -> bool:
    """Verdadero si misma clase y mm.IoU(caja_pred, caja_real) >= umbral."""
    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:
    """Convierte la predicción de Roboflow a (clase, (x, y, w, h))."""
    caixa = (p["x"] - p["width"] / 2, p["y"] - p["height"] / 2,
             p["width"], p["height"])
    # usa el nombre de la clase (no el class_id!) para hacer coincidir con el orden local
    classe = CLASSES_LOCAIS.index(p["class"])
    return (classe, caixa)

def carrega_labels_yolo(caminho_txt: str, largura: int, altura: int) -> list:
    """Lee etiquetas YOLO (normalizadas) y convierte a (clase, (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("Clave de Roboflow no encontrada: chave_roboflow.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"{acertos}/{len(caixas_pred)} predicciones correctas (clase + IoU >= 50%)")
```

Para visualizar el resultado, cada predicción se dibuja sobre la imagen: en verde cuando es un acierto (clase + IoU umbral) y en rojo cuando es un error, como se muestra en Figura 9.40.

```
import cv2

VERDE, VERMELHO = (0, 255, 0), (255, 0, 0)

if not chave.exists():
    print("Clave de Roboflow no encontrada: chave_roboflow.txt")
else:
    img_acertos = mm.read(img_path).copy()

    if not chave.exists():
        print("Clave de Roboflow no encontrada: 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)} predições corretas "
            f"(classe + IoU >= 50%)",
        figsize=(6, 6)
    )
```

Figura 9.40

9.5.5 Aplicaciones Geométricas e Integradas

El capítulo concluye integrando dos pilares de la VC: la **geometría proyectiva** (estudiada en los Capítulos 6 y 8) y el **aprendizaje profundo**. La combinación de estos enfoques sustenta aplicaciones prácticas en el mundo real, como se ilustra a continuación.

9.5.5.1 Realidad Aumentada con Marcadores y Homografía

Imagina una cámara apuntando a una mesa donde alguien pegó un pequeño marcador ArUco. Dependiendo del ángulo de la cámara, ese marcador aparece **rotado, inclinado, en perspectiva** — nunca perfectamente cuadrado. Es precisamente esa distorsión la que la homografía sabe "leer" y deshacer (o, en nuestro caso, replicar para una nueva imagen).

El flujo completo de una aplicación de RA basada en marcadores sigue tres pasos:

1. **Ambientación:** el marcador se inserta en una escena real, sufriendo una transformación de perspectiva (simulando el ángulo de la cámara).
2. **Detección:** el algoritmo localiza el marcador en la escena y recupera las coordenadas exactas de sus 4 esquinas con `cv2.aruco.ArucoDetector`.
3. **Sustitución:** con la homografía entre el marcador "ideal" y el marcador "detectado", se proyecta una **nueva imagen virtual** exactamente sobre el área del marcador — como si este se hubiera transformado en una ventana hacia otro contenido, tal como muestra la Figura 9.41.

> **Nota**
>
> **Detalle técnico importante:** el ArUco necesita un margen blanco alrededor del patrón (la "zona de silencio") para que el detector pueda diferenciar el marcador del fondo. Por ello, el código siguiente añade un borde con `cv2.copyMakeBorder` y utiliza interpolación `cv2.INTER_NEAREST` al deformar la imagen, evitando que la rotación difumine los cuadraditos blancos y negros e impida la detección.

```
# 1. Generación del marcador ArUco sintético, ya con margen blanco (zona de silencio)
# → este margen es esencial para que el detector pueda "ver" el marcador
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, con 40px de marco blanco en cada lado
aruco = cv2.cvtColor(aruco_gray, cv2.COLOR_GRAY2BGR)
lado = aruco.shape[0]  # 280

# 2. Escena real de fondo, usando una imagen de ejemplo del skimage.data
fundo = cv2.cvtColor(skdata.coffee(), cv2.COLOR_RGB2BGR)
cena = cv2.resize(fundo, (640, 480))

# Esquinas del marcador "de frente" (src) y su posición rotada/inclinada en la escena (dst)
src = np.float32([[0, 0], [lado, 0], [lado, lado], [0, lado]])
dst = np.float32([[190, 160], [420, 70], [470, 330], [150, 370]])  # rotación + perspectiva

# 3. Proyecta el marcador (con bordes nítidos) sobre la escena real
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. Detección del marcador dentro de la escena (como lo haría una cámara)
```

```
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. Imagen virtual (otra imagen del skimage.data) que va a "sustituir" al marcador
# Nota: usamos el cuadrado INTERNO del marcador (sin el margen) como área de proyección,
# entonces la homografía de la imagen virtual usa 'src' original (200x200), no el 'lado' con
borde
src_interno = np.float32([[0, 0], [200, 0], [200, 200], [0, 200]])
virtual = cv2.resize(
    cv2.cvtColor(skdata.camera(), cv2.COLOR_RGB2BGR), (200, 200)
)

# Las esquinas detectadas corresponden al marcador CON el margen (280x280),
# entonces recalculamos H_ra usando 'src' con margen, para mantener la proyección coherente
H_ra, _ = cv2.findHomography(src, corners[0][0])

# La homografía detectada se aplica a la imagen virtual (redimensionada a 'lado')
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]

# Visualización: escena con marcador vs. escena con Realidad Aumentada aplicada
mm.show(
    [cv2.cvtColor(cena, cv2.COLOR_BGR2RGB), cv2.cvtColor(ra, cv2.COLOR_BGR2RGB)],
    titles=["Marcador en la Escena Real (rotado)", "Superposición Virtual mediante
Homografía"],
    cols=2,
    figsize=(9, 4),
)
```

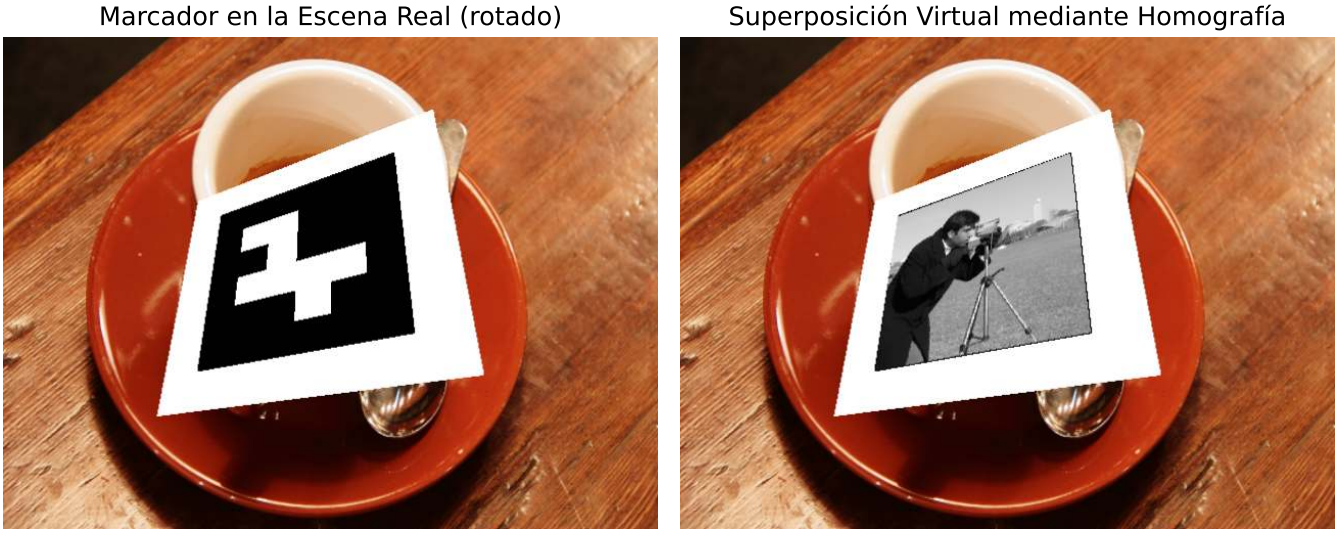


Figura 9.41: Realidad aumentada basada en marcador ArUco: marcador insertado en escena real y rotado, detectado y sustituido por imagen virtual mediante homografía.

Resumen del *pipeline*:

Etapa	Qué hace	Función clave
1. Generación	Marcador ArUco con margen blanco	`generateImageMarker` + `copyMakeBorder`
2. Ambientación	Inserta marcador distorsionado en la escena	`findHomography` + `warpPerspective`
3. Detección	Localiza marcador y devuelve esquinas	`ArucoDetector.detectMarkers`
4. Sustitución	Proyecta imagen virtual sobre el marcador	`findHomography` + `warpPerspective`

💡 **Lección:** que el detector "no encuentre nada" es común en VC. Pregúntate siempre: **"¿di contraste y espacio suficientes?"** — vale para ArUco, códigos QR y reconocimiento facial.

9.5.5.2 Fotogrametría y Referencia de Escala

La **fotogrametría** permite estimar dimensiones físicas de objetos a partir de imágenes digitales. Para ello, se utiliza un objeto de referencia con dimensiones conocidas, posicionado en la misma escena que el objeto de interés. Este procedimiento establece una relación entre distancias medidas en *píxeles* y sus correspondientes dimensiones en el mundo real.

Considere, por ejemplo, una tarjeta de crédito, cuyas dimensiones siguen el estándar internacional ISO/IEC 7810. Como su ancho es exactamente **8,56 cm**, basta con determinar cuántos *píxeles* ocupa ese ancho en la imagen para calcular el factor de conversión entre *píxeles* y centímetros. Si la tarjeta corresponde a 140 *píxeles*, entonces cada *píxel* representará aproximadamente **0,061 cm**. Este mismo factor de escala puede aplicarse para estimar las dimensiones de cualquier otro objeto ubicado en el mismo plano de la escena, tal como se ilustra en la Figura 9.42.

El procedimiento puede dividirse en dos etapas principales:

1. **Segmentación y *bounding box*:** localizar, en la imagen, tanto el objeto de referencia como el objeto de interés, utilizando técnicas como segmentación por color, umbralización, detección de contornos o métodos de detección de objetos.
2. **Conversión a dimensiones físicas:** calcular la razón cm/pixel a partir del ancho conocido del objeto de referencia y utilizarla para convertir las medidas del objeto de interés de *píxeles* a centímetros.

> **Nota**
>
> **Condición para mediciones confiables**
> La conversión entre *píxeles* y centímetros presupone que el objeto de referencia y el objeto de interés se encuentren aproximadamente en el mismo plano y a la misma distancia de la cámara. Bajo estas condiciones, la escala permanece prácticamente constante en toda la imagen. Diferencias de profundidad, inclinación de la cámara o distorsiones de la lente pueden introducir errores en las medidas estimadas.

```
COR_REFERENCIA = (200, 200, 200)  # Tarjeta de referencia (gris)
COR_OBJETO = (60, 60, 220)  # Objeto objetivo (rojo)

# Dibujo de la escena sintética
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)
```

```python
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
)

# Cálculo de escala física
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

# Dibujo de las cajas y mediciones estimadas
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=[
        "Escena Original",
        "Medición por Referencia de Escala (Centímetros)",
    ],
```

```
    cols=2,
    figsize=(9, 4),
)
```

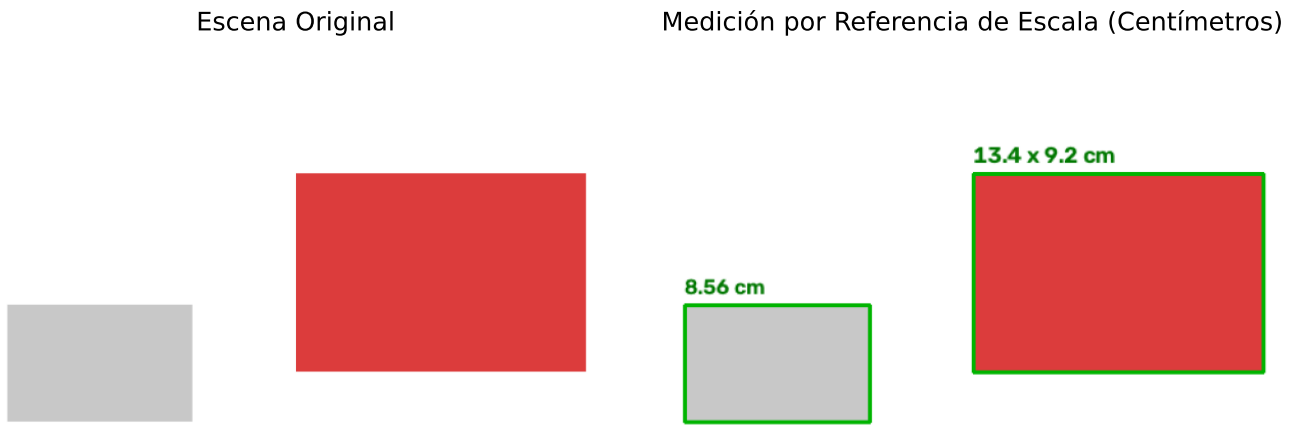


Figura 9.42: Medición de dimensiones físicas reales utilizando una tarjeta de referencia de escala conocida (8,56 cm) renderizada. El rectángulo gris representa la tarjeta de referencia y el rectángulo azul representa el objeto objetivo. Las dimensiones estimadas del objeto se muestran en centímetros.

Resumen del *pipeline*:

Etapa	Qué hace	Función clave
1. Escena sintética	Dibuja la tarjeta de referencia y el objeto objetivo con colores distintos	`cv2.rectangle`
2. Segmentación	Aísla cada objeto por color y extrae su contorno	`cv2.findContours`
3. Bounding Box	Obtiene la caja delimitadora (posición y tamaño en píxeles) de cada objeto	`cv2.boundingRect`
4. Escalado	Convierte píxeles en centímetros usando el ancho conocido de la tarjeta	Regla de tres: $\text{cm/pixel} = \frac{8{,}56}{w_{ref_px}}$
5. Anotación	Dibuja las cajas y muestra las medidas estimadas sobre la imagen	`cv2.rectangle` + `cv2.putText`

💡 **Aplicación en el mundo real:** esta es exactamente la técnica utilizada por *apps* de *e-commerce* que estiman el tamaño de un producto a partir de una foto junto a una tarjeta, por sistemas agrícolas que miden frutas en cintas transportadoras, e incluso por pericias forenses que calculan dimensiones de vestigios en escenas del crimen — todo con la misma idea: **una regla conocida dentro de la propia foto.**

9.6 Resumen

Este capítulo, que cierra la Parte II del libro, presentó:

- **Convolución y *pooling* aprendidos:** la misma operación matemática de convolución del Capítulo 3, pero con *kernels* tratados como parámetros ajustados mediante entrenamiento, en lugar de definidos manualmente;

- **Compartición de pesos y jerarquía de características** como propiedades que hacen que las CNN sean eficientes y capaces de aprender representaciones cada vez más abstractas en capas sucesivas;
- **Entrenamiento de una CNN desde cero**, con un rendimiento comparable — y no necesariamente superior — al de los clasificadores clásicos del Capítulo 7 en una base pequeña y simple, reforzando que la elección del método debe ser proporcional a la complejidad real del problema;
- **Transferencia de aprendizaje**, demostrada experimentalmente como una estrategia eficaz para tareas con pocos datos etiquetados, reutilizando un extractor de características ya entrenado en una tarea relacionada, y sus limitaciones, evidenciadas por la transferencia negativa entre dominios muy distintos;
- **Aplicaciones a gran escala** con modelos preentrenados de clasificación, detección (*Faster R-CNN*) y segmentación (*DeepLabV3*), siguiendo el mismo principio de transferencia de aprendizaje a escala industrial;
- **Transferencia de aprendizaje aplicada a la detección de objetos**, ajustando un YOLO preentrenado en COCO para localizar y clasificar objetos geométricos sintéticos, ilustrando el mismo principio de congelamiento parcial en un dominio de origen y destino aún más distantes entre sí;
- **Segmentación semántica con U-Net**, implementada y entrenada desde cero sobre un conjunto sintético de bajo contraste, superando una línea de base clásica de umbralización gracias a las conexiones de atajo entre codificador y decodificador;
- **Ingeniería de datos para Visión por Computador con *Roboflow***, utilizando un *dataset* y un modelo preentrenado para realizar inferencia en la detección de objetos geométricos, además de comparar conjuntos de datos con diferentes resoluciones, pero con las mismas clases de objetos;
- La integración de **geometría computacional** (homografía y calibración) y **aprendizaje profundo** en dos aplicaciones reales que cierran el libro: **realidad aumentada** y **fotogrametría**.

9.7 🤖 Uso de Gemini Notebook como Tutor Complementario

En esta edición, se incentiva el uso de **Gemini Notebook** como herramienta complementaria de aprendizaje. Basado en inteligencia artificial, el sistema utiliza exclusivamente los documentos proporcionados por el autor como fuente de conocimiento, produciendo respuestas alineadas con el contenido y el enfoque adoptado a lo largo de este capítulo.

🎓 Estudia con el Tutor Inteligente

🚀 ACCEDER A GEMINI NOTEBOOK: CAPÍTULO 09

🌐 Idioma y Lenguaje de Programación

El proyecto de este capítulo en Gemini Notebook se construyó únicamente con el texto en **portugués** y los ejemplos de código en **Python**. Si estás estudiando con la edición en inglés o francés, o siguiendo la ruta en C++, las respuestas del tutor pueden no corresponder exactamente con la versión que estás leyendo.

⚠️ Aviso sobre Contenido Generado por IA

Aunque es una herramienta valiosa de apoyo al estudio, Gemini Notebook puede eventualmente producir respuestas incompletas, imprecisas o incorrectas. Se recomienda validar la información consultando el material del capítulo, libros, artículos científicos y otras fuentes académicas confiables. Siempre que sea posible, ejecuta y experimenta con los ejemplos prácticos presentados a lo largo del texto para consolidar la comprensión de los conceptos.

9.8 Lista de Ejercicios

Los ejercicios siguientes consolidan los conceptos presentados en este capítulo mediante adaptaciones, experimentos y extensiones de los algoritmos desarrollados a lo largo del texto, utilizando las bibliotecas `PyTorch`, `ultralytics` y la biblioteca didáctica `morph`.

1. **(10%)** Investigue el impacto de la profundidad en una arquitectura convolucional. Partiendo de la red de dos capas del Proyecto Práctico 1, añada una tercera capa convolucional con 32 filtros antes de las capas totalmente conectadas. Entrene la nueva arquitectura manteniendo el mismo número de épocas y la misma división de los datos. Compare la exactitud en el conjunto de prueba y el número total de parámetros entrenables en relación con la red original, discutiendo si el aumento de profundidad aportó un beneficio medible para imágenes de dimensión 8×8.

2. **(15%)** Evalúe el umbral de datos necesarios en el dominio de destino para que el entrenamiento de una CNN desde cero se vuelva competitivo con la **transferencia de aprendizaje**. Variando el número de muestras de entrenamiento disponibles en el dominio B para $\{5, 10, 20, 40, 80\}$, mida la exactitud de prueba para ambas estrategias. Presente los resultados en un gráfico de líneas y determine a partir de qué volumen de datos el entrenamiento desde cero alcanza un rendimiento equivalente al extractor preentrenado.

3. **(15%)** Investigue la estrategia de **ajuste fino parcial** (*fine-tuning*) en comparación con el congelamiento total de pesos. En el escenario de transferencia de aprendizaje entre dominios de dígitos, descongele la segunda capa convolucional (`conv2`) del extractor para que se actualice junto con la cabeza de clasificación durante el entrenamiento en el dominio B. Compare la exactitud obtenida con el congelamiento total y con el entrenamiento desde cero, discutiendo el compromiso entre capacidad de adaptación y riesgo de sobreajuste (*overfitting*).

4. **(20%)** Evalúe la influencia de la profundidad del congelamiento (*freeze*) en el rendimiento de detectores **YOLO** sometidos a transferencia de aprendizaje. Utilizando el conjunto de datos sintético de formas geométricas, ejecute el ajuste fino variando el parámetro de congelamiento del *backbone* para $\{0, 5, 10, 15\}$. Registre la métrica mAP_{50} en el conjunto de validación para cada configuración, presente los datos en una tabla y discuta si el congelamiento parcial es ventajoso cuando los dominios de origen (COCO) y de destino (formas geométricas) son significativamente distintos.

5. **(20%)** Estudie la importancia de las **conexiones de atajo** (*skip connections*) en la arquitectura **U-Net** para segmentación semántica. Implemente una variación `UNetSemAtalhos` que funcione como un *autoencoder* convolucional tradicional, eliminando las concatenaciones entre las etapas del codificador y del decodificador. Entrene ambos modelos sobre la misma base de nódulos sintéticos, compare el IoU medio en el conjunto de validación y presente visualmente la diferencia en la precisión de los bordes segmentados por cada método.

6. **(20%) — Desafío: segmentación semántica con *Roboflow*.** Utilice un proyecto de *Roboflow* del tipo *instance segmentation*, que contenga las siete clases de formas geométricas utilizadas en este capítulo. Exporte el *dataset* en el formato `coco-segmentation` y desarrolle un procedimiento para convertir los polígonos almacenados en los archivos `_annotations.coco.json` en máscaras semánticas multiclase, en las cuales cada píxel recibe el índice de la clase correspondiente y el valor `0` representa el fondo. Utilice las imágenes y máscaras resultantes para entrenar la `UNetCompacta`. Evalúe el IoU medio en el conjunto de prueba y compare visualmente las máscaras predichas con las anotaciones originales. Discuta las principales dificultades encontradas en la conversión de las anotaciones COCO a máscaras y los efectos de objetos superpuestos o pertenecientes a diferentes clases.

7. **(Bono – 10%)** Desarrolle un sistema interactivo que combine **detección de objetos (YOLO)** con **medición por referencia de escala (fotogrametría)**. Entrene el detector para identificar dos clases en una escena: una "Tarjeta de Referencia" (dimensión conocida de 8,56 cm $\times$ 5,39 cm) y un "Objeto Objetivo". Al realizar la inferencia en una nueva imagen, utilice la dimensión en píxeles de la *bounding box* de la tarjeta detectada para convertir las dimensiones de la caja del objeto objetivo en centímetros. Muestre la imagen procesada con las etiquetas de clase, la probabilidad de confianza y las dimensiones

físicas estimadas superpuestas.

8. **(Bono – 10%)** Desarrolle un sistema de **estimación de profundidad por visión estéreo** a partir de dos imágenes de la misma escena obtenidas desde posiciones diferentes, simulando un par de cámaras estéreo. Considere que la distancia entre las dos posiciones de captura (*baseline*) sea conocida.

   Utilice uno de los métodos de **detección de objetos** presentados en el capítulo, como YOLO, para localizar los objetos de interés en las dos imágenes. Para cada detección, establezca la correspondencia entre el mismo objeto en las dos posiciones y determine su **disparidad**. A partir de la disparidad, del *baseline* y de los parámetros de la cámara, utilice la geometría estéreo para estimar la distancia de cada objeto respecto a las cámaras.

   Como extensión de la biblioteca didáctica `morph`, modifique el método `showBoundBox` para que, además de la **clase** y la **confianza de la detección**, presente sobre cada *bounding box* la **distancia estimada del objeto**. El resultado debe permitir visualizar, directamente en las imágenes, la clase, la exactitud (confianza) y la profundidad de cada objeto detectado.

   Presente las dos imágenes con las detecciones, las correspondencias entre los objetos, la imagen de disparidad y una representación de la profundidad estimada. Discuta cómo la distancia entre las cámaras, la precisión de la detección y de la correspondencia, la resolución de las imágenes y la posición del objeto en la escena influyen en la calidad de la estimación.

   Para la validación, utilice al menos un objeto cuya distancia a la cámara sea conocida. Compare la profundidad estimada con el valor real y reporte el **error absoluto** y el **error relativo**. Discuta también las limitaciones del método cuando un objeto no se detecta correctamente en las dos imágenes o cuando la correspondencia entre las regiones observadas es ambigua.

9.9 Cierre de la Parte II

Este capítulo concluye la Parte II del libro y cierra la secuencia de contenidos iniciada en el Capítulo 6, dedicada a la representación, detección, descripción y correspondencia de características en imágenes. A lo largo de estos capítulos, se presentaron métodos clásicos de VC basados en **características diseñadas manualmente**, como Sobel, LBP, HOG, ORB y Haar Cascade, así como métodos fundamentados en **características aprendidas automáticamente**, representados por las CNN.

Los ejemplos y experimentos desarrollados evidencian que ninguna de estas aproximaciones es universalmente superior. La elección de la técnica más adecuada depende de las características del problema, de la disponibilidad de datos para entrenamiento, de los requisitos de precisión y de las restricciones computacionales de la aplicación. En problemas bien estructurados y con pocos datos, los descriptores clásicos frecuentemente ofrecen soluciones simples y eficientes. En contrapartida, las tareas más complejas tienden a beneficiarse de la capacidad de aprendizaje proporcionada por las CNN.

Diversas direcciones de estudio pueden profundizar los conceptos presentados en esta parte del libro, entre las cuales se destacan:

- **Arquitecturas modernas de CNN**, como ResNet, EfficientNet y *Vision Transformers*, que amplían la capacidad de representación y el rendimiento en tareas de clasificación y reconocimiento visual;
- **Detección y segmentación de objetos**, con especial atención a las familias YOLO y a los modelos de segmentación basados en *prompts*, como el *Segment Anything*;
- **Reconstrucción tridimensional y SLAM** (*Simultaneous Localization and Mapping*), que utilizan múltiples imágenes para estimar la geometría de la escena y la trayectoria de cámaras en movimiento;
- **Modelos generativos de imágenes**, como redes adversarias generativas (GAN) y modelos de difusión, capaces de sintetizar imágenes realistas a partir de ejemplos o descripciones textuales.

Los fundamentos desarrollados a lo largo de la Parte II constituyen la base para estas y otras áreas avanzadas de la VC, en las cuales la representación adecuada de la información visual permanece como elemento central para el análisis y la comprensión de imágenes.

Referencias del Capítulo

Los conceptos y algoritmos presentados en este capítulo se fundamentaron en referencias clásicas y contemporáneas de la literatura de Aprendizaje Profundo aplicado a la VC:

- Mcculloch (1943), para la propuesta de la primera abstracción matemática y lógica del neurona artificial, fundamentando las bases conceptuales del procesamiento neural computacional.
- Rosenblatt (1958), para la formulación original del **Perceptrón**, modelo precursor del neurona artificial utilizado en las arquitecturas modernas de aprendizaje profundo.
- Goodfellow (2016) y Lecun (2015), para los fundamentos de redes neuronales, convolución, funciones de activación y entrenamiento de modelos profundos.
- Bishop (2006), para los conceptos rigurosos de reconocimiento de patrones, probabilidad, estimación de máxima verosimilitud y métodos estadísticos aplicados al aprendizaje automático.
- Ronneberger (2015), para la arquitectura **U-Net**, utilizada en la segmentación semántica con conexiones de atajo entre codificador y decodificador.
- Redmon (2016), para la arquitectura **YOLO** (*You Only Look Once*), utilizada en los experimentos de detección de objetos con transferencia de aprendizaje.
- Ren (2015), para la arquitectura **Faster R-CNN**, empleada como modelo preentrenado de detección de objetos.
- He (2016), para la arquitectura **ResNet**, base de diversos extractores de características preentrenados utilizados en este capítulo.
- Chen (2018), para la arquitectura **DeepLabV3**, utilizada como modelo preentrenado de segmentación semántica.
- Kirillov (2023), para el modelo **Segment Anything (SAM)**, mencionado como dirección de estudio para segmentación promptable.
- {google} (2025), referente a la herramienta **Gemini Notebook**, utilizada en la elaboración del infográfico de síntesis del capítulo y disponible como apoyo complementario al estudio.

Executar Colab | Abrir GitHub

9.10 💻 Parte Práctica con Ejercicios de Programación

La presente lista de **Ejercicios de Programación (EP)** consolida las formulaciones teóricas presentadas a lo largo del Capítulo 9 — Aprendizaje Profundo para Visión por Computador — mediante una ruta práctica aplicada. A diferencia del entrenamiento de redes neuronales completas con PyTorch, que requiere tiempo de ejecución y, en ocasiones, GPU, los EP de este capítulo aislan las **magnitudes intermedias** de un *pipeline* real de aprendizaje profundo —la salida de una única capa convolucional, el resultado de una operación de *pooling*, el conteo de parámetros entrenables de una arquitectura, la superposición entre cajas delimitadoras candidatas, la calidad de una máscara de segmentación y el filtro de supresión de no máximos—, permitiendo validar manualmente cada etapa del razonamiento sin depender de bibliotecas de aprendizaje automático ni de entrenamiento real.

El encadenamiento de los ejercicios reproduce el flujo conceptual del capítulo y crece en dificultad a cada paso: se comienza con el cálculo manual de la salida de una **capa convolucional aprendida** (🟢), a partir de un *kernel* y un sesgo ya entrenados; se avanza hacia la operación de ***pooling*** (🟢, máximo y promedio), que reduce la resolución espacial entre bloques convolucionales; se continúa con el **conteo de parámetros entrenables** (🟡) de una arquitectura CNN completa, evidenciando por qué el compartimiento de pesos vuelve a estas redes mucho más económicas que una capa totalmente conectada equivalente; se profundiza en el cálculo de **Intersección sobre Unión (IoU)** y en la **Supresión de No Máximos (NMS)** (🟡), etapa de posprocesamiento común a detectores como Faster R-CNN y YOLO; se sigue con la **evaluación de máscaras de segmentación** (🟠) con las mismas métricas de IoU y Dice usadas para comparar la U-Net con la línea base morfológica clásica; y se concluye con un ***pipeline* integrado** (🔴), que une la salida de un detector de objetos (después de NMS) con una medición del mundo real por referencia de escala —el mismo

principio de la fotogrametría estudiada en la integración final del capítulo.

Siempre que tenga sentido, cada ejercicio señala métodos de la biblioteca didáctica `morph.py` (la misma usada a lo largo del capítulo, importada como `mm`) que resuelven una etapa del problema o que sirven de referencia para verificar tus propios cálculos —sin, no obstante, sustituir el razonamiento que debes implementar.

❗ Directrices para la Resolución de los Ejercicios de Programación

En todos los ejercicios de este capítulo, las etapas de discretización o redondeo numérico deben emplear el redondeo estándar al entero más cercano (*round half away from zero*), mitigando ambigüedades en valores con fracción exactamente igual a 0,5. Salvo indicación explícita en contrario: (i) la operación de "convolución" sigue la convención adoptada por los *frameworks* de aprendizaje profundo —**correlación cruzada**, sin inversión espacial del *kernel*, exactamente como se presentó en la Sección "Capa Convolucional"; (ii) el relleno (*padding*) se realiza con ceros; (iii) las cajas delimitadoras se especifican en el formato esquina-a-esquina (x_1, y_1, x_2, y_2), con $x_1 < x_2$ e $y_1 < y_2$; y (iv) los vectores/matrices siguen indexación desde 0, con la convención `[fila][columna]` para estructuras bidimensionales.

🎯 Objetivo de este Cuaderno

El cuaderno permite desarrollar, validar, organizar y probar soluciones de **Ejercicios de Programación (EPs)** en entornos interactivos, como Colab, con los mismos casos de prueba de Moodle, copiándolos allí solo al momento de registrar la nota oficial.

Descargar

Descargue `morph.py` y `testsuite.py` ejecutando la celda a continuación:

```
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
```

```
Entorno listo. Morph: 1.1.9 | OpenCV: 5.0.0 | TestSuite: 1.1.2
```

Ejecutando los Tests

Para evaluar los tests, ejecute `TestSuite("EP09_01.extensión").run()` en una nueva celda, reemplazando la extensión por la del lenguaje utilizado (`.py`, `.java`, `.c`, `.cpp`, `.js` o `.r`). El sistema descarga los casos de prueba desde GitHub, ejecuta el programa y calcula la nota automáticamente.

Para probar código Python directamente, sin guardar archivo, use `run_code(codigo)` pasando el código como *string* en una variable `codigo`:

```
codigo = """
# ... su código aquí ...
"""
TestSuite("EP09_01").run_code(codigo)
```

9.10.1 EP09_01 🟢 Convolución 2D Manual (*Forward* de una Capa Aprendida)

PyTorch, presentado en este capítulo, ejecuta `nn.Conv2d(x)` en una única llamada — pero detrás de ella solo está la correlación cruzada entre un *kernel* (ya entrenado) y una vecindad de la entrada, seguida de la suma de un sesgo y de una activación, exactamente como se formalizó en la Sección "Capa Convolucional". La diferencia esencial respecto a la convolución de *kernels* fijos del Capítulo 3 es que, aquí, los valores del *kernel* y del sesgo **ya vienen listos** (como si hubieran sido aprendidos por gradiente), y te corresponde a ti reproducir manualmente la pasada directa (*forward pass*) que el *framework* ejecuta internamente.

Antes de entrenar una CNN real, se te ha encargado implementar esta pasada directa desde cero, para una única capa convolucional con un único canal de entrada y un único filtro de salida, incluyendo soporte para *padding* y *stride* arbitrarios.

9.10.1.1 📋 Directrices de Implementación

1. **Entrada:** Leer las dimensiones $H \times W$ del mapa de características de entrada y, a continuación, sus $H \times W$ valores reales.
2. ***Kernel* y sesgo:** Leer las dimensiones $k_h \times k_w$ del *kernel* (ya entrenado), sus valores reales, y el sesgo b (real, escalar).
3. **Hiperparámetros:** Leer el *padding* p (entero, número de ceros añadidos en cada borde) y el *stride* s (entero, paso del deslizamiento).
4. **Relleno:** Añadir p ceros en cada uno de los cuatro bordes del mapa de entrada antes de la correlación.
5. **Correlación cruzada:** Para cada posición de salida (i, j), calcular

   $$z(i,j) = b + \sum_{u=0}^{k_h-1} \sum_{v=0}^{k_w-1} K(u,v) \cdot X_{pad}(i \cdot s + u,\ j \cdot s + v),$$

   recorriendo la entrada **sin** invertir el *kernel* (convención de los *frameworks* de aprendizaje profundo, diferente de la convolución matemática clásica).
6. **Activación:** Aplicar ReLU a cada valor: $a(i,j) = \max(0, z(i,j))$.
7. **Dimensiones de salida:** $O_h = \lfloor (H + 2p - k_h)/s \rfloor + 1$ y $O_w = \lfloor (W + 2p - k_w)/s \rfloor + 1$.
8. **Salida:** Imprimir O_h y O_w en la primera línea, seguidos de O_h líneas con O_w valores reales cada una (el mapa de características de salida, ya con ReLU aplicada), formateados con 4 decimales.

9.10.1.2 📌 Restricciones Computacionales

- **Un canal de entrada, un filtro de salida:** no es necesario manejar múltiples canales ni múltiples filtros en esta versión simplificada.
- **Sin inversión del *kernel*:** implementa correlación cruzada, no la convolución matemática clásica con *kernel* invertido — es esa la operación que PyTorch (y la mayoría de los *frameworks*) llama "convolución".
- **Relleno con ceros:** los p píxeles añadidos en cada borde valen siempre 0.
- **Formato:** todos los valores de salida deben tener exactamente 4 decimales, incluso cuando el valor es entero (ej.: `2.0000`).

9.10.1.3 🧠 Fundamentación Teórica

Elemento	Papel en la capa convolucional
Kernel K	Parámetros aprendidos por gradiente, análogos a los coeficientes de un filtro fijo del Capítulo 3, pero ajustados por retropropagación

Elemento	Papel en la capa convolucional
Sesgo b	Desplazamiento aprendido, sumado tras la correlación — permite que la neurona "se active" incluso con entrada nula
Padding	Controla la dimensión espacial de salida y evita la pérdida de información en los bordes en cada capa
Stride	Controla el paso del desplazamiento; valores > 1 reducen la resolución espacial, como una forma de submuestreo integrado en la propia convolución
ReLU	Introduce no linealidad tras la combinación lineal, exactamente como en la Sección "Función de Activación"

9.10.1.4 🧩 Métodos del `morph.py` que pueden ayudar

- `mm.readImg(h, w, dtype='float')` — lee directamente una matriz $h \times w$ de valores reales de la entrada estándar, ahorrando el *parseo* manual del mapa de características y del *kernel*.
- `mm.correlacao0(f, kernel, bias)` — implementa la misma suma de correlación cruzada + sesgo que vas a calcular a mano, pero **sin** soporte para *padding* o *stride*, y convierte el resultado a `uint8` (trunca valores negativos y decimales). Puede servir como referencia conceptual o para comprobar el caso más simple ($p = 0$, $s = 1$), pero no sustituye tu implementación completa — que debe preservar signo, decimales, *padding*, *stride* y ReLU.

9.10.1.5 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Enteros H y W.
- Siguientes H líneas: W valores reales cada una (mapa de entrada).
- Siguiente línea: Enteros k_h y k_w.
- Siguientes k_h líneas: k_w valores reales cada una (*kernel*).
- Siguiente línea: Real b (sesgo).
- Siguiente línea: Enteros p y s.

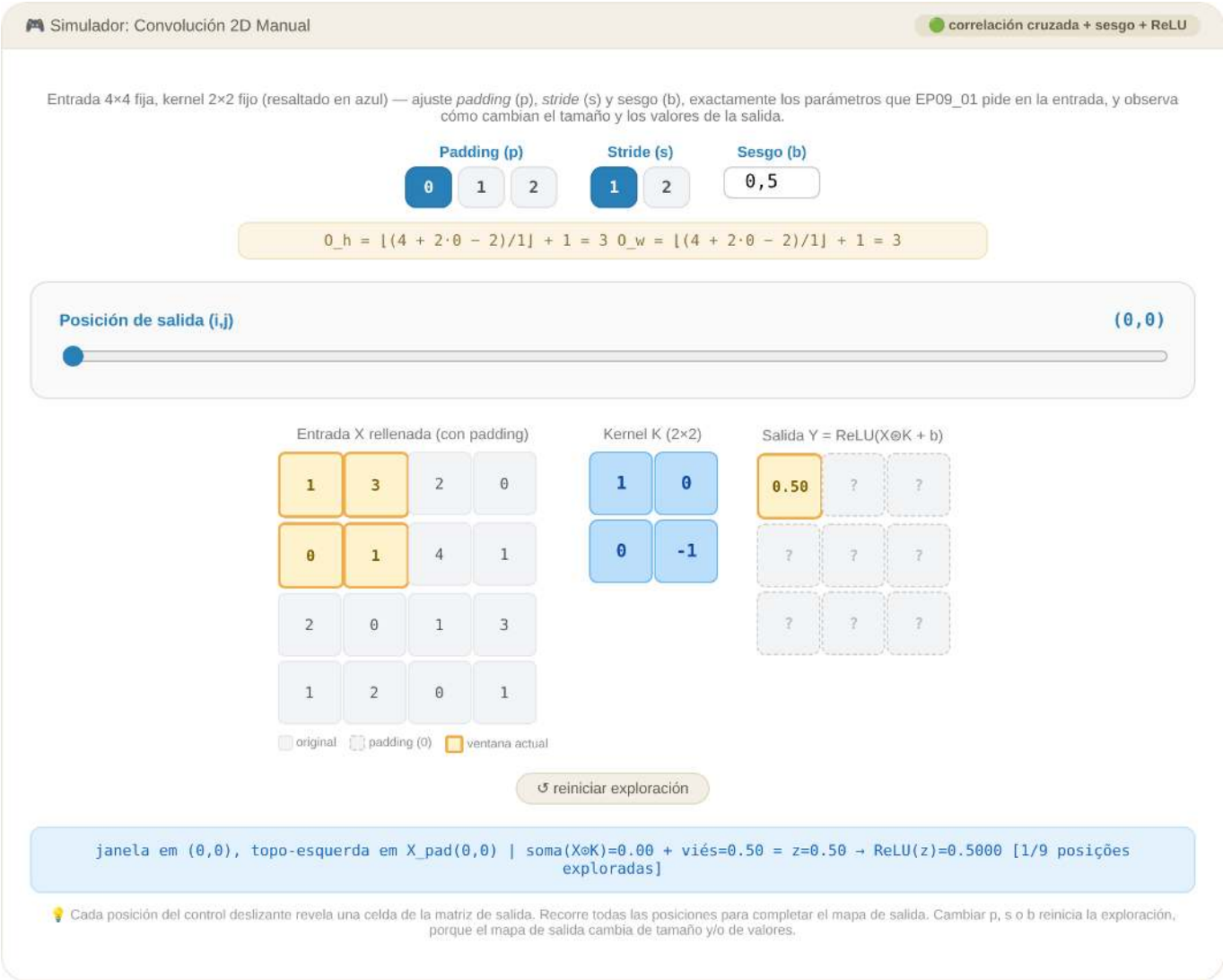
Salida:

- Línea 1: Enteros O_h y O_w.
- Siguientes O_h líneas: O_w valores reales cada una, con 4 decimales.

9.10.1.6 📌 Ejemplos

Entrada	Salida	Observación
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: salida 2×2 sin relleno.

Entrada	Salida	Observación
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: entrada rellenada con ceros antes de la correlación.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-ep0901.html

Figura 9.43: Simulador EP09_01: Convolución 2D Manual (correlación cruzada + sesgo + ReLU, con *padding* y *stride* ajustables)

```
%%writefile EP09_01.py
# Código Python
```

```
Overwriting EP09_01.py
```

```
TestSuite("EP09_01.py").run()
```

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

9.10.2 EP09_02 🟢 *Pooling* Manual (Máximo y Media)

Entre bloques convolucionales, la arquitectura típica de una CNN intercala capas de ***pooling***, que reducen la resolución espacial del mapa de características sin introducir nuevos parámetros entrenables — a diferencia de la convolución, el *pooling* no tiene pesos: solo resume cada ventana de la entrada a un único valor, mediante un máximo o una media, exactamente como se formalizó en la Sección "*Pooling*".

Se le ha encargado implementar esta operación a partir de una ventana deslizante cuadrada, sin superposición parcial en los bordes (solo ventanas completas), soportando los dos tipos más comunes: `max` (preserva el valor más sobresaliente, típicamente usado para retener bordes y texturas fuertes) y `avg` (suaviza la región, preservando información de intensidad media).

9.10.2.1 📋 Directrices de Implementación

1. **Entrada:** Leer las dimensiones $H \times W$ del mapa de características de entrada y sus $H \times W$ valores reales.
2. **Ventana:** Leer los enteros k (tamaño de la ventana cuadrada $k \times k$) y s (*stride*).
3. **Tipo:** Leer una *string*, `max` o `avg`, indicando el tipo de *pooling*.
4. **Sin relleno:** Esta operación **no** utiliza *padding*; las ventanas que sobrepasarían el borde de la entrada se descartan.
5. **Cálculo:** Para cada posición de salida (i, j), calcular el máximo o la media de los $k \times k$ valores de la ventana correspondiente, comenzando en $(i \cdot s,\ j \cdot s)$.
6. **Dimensiones de salida:** $O_h = \lfloor (H - k)/s \rfloor + 1$ y $O_w = \lfloor (W - k)/s \rfloor + 1$.
7. **Salida:** Imprimir O_h y O_w en la primera línea, seguidos de O_h líneas con O_w valores reales cada una, formateados con 4 decimales.

9.10.2.2 📌 Restricciones Computacionales

- **Ventana cuadrada:** $k \times k$, sin soporte para ventanas rectangulares en esta versión.
- **Sin *padding*:** solo se consideran ventanas completamente contenidas en la entrada — las dimensiones que "sobran" simplemente se descartan.
- **`avg` usa división real:** la media es siempre suma/k^2, incluso cuando el resultado tiene muchos decimales — redondee solo en el formato final, conforme a la directriz general del capítulo.
- **Formato:** todos los valores de salida con exactamente 4 decimales.

9.10.2.3 🧠 Fundamentación Teórica

Elemento	Rol en la arquitectura
Pooling máximo	Preserva la activación más fuerte de la ventana; común después de capas convolucionales para retener bordes y texturas sobresalientes
Pooling medio	Suaviza la región, preservando la intensidad media; común en capas finales (*global average pooling*)
Ausencia de parámetros	Diferencia el *pooling* de la convolución: reduce la resolución espacial sin costo adicional de entrenamiento
Reducción de resolución	Contribuye a la invariancia a pequeñas traslaciones y a la reducción del costo computacional de las capas siguientes

9.10.2.4 🧩 Métodos de `morph.py` que pueden ayudar

El `morph.py` no implementa *pooling* con submuestreo directamente, pero dos familias de operaciones muestran la misma idea desde otra óptica, útil para verificar su intuición:

- `mm.dil(f, Bc)` / `mm.dil0(f, B)` — dilatación morfológica: reemplaza cada píxel por el **máximo** de su vecindad definida por el elemento estructurante B (ej.: `mm.sebox(n)` para una ventana $(2n+1)\times(2n+1)$). Es, conceptualmente, un "*max-pooling* sin submuestreo" (produce una imagen del mismo tamaño, en lugar de reducida).
- `mm.blur(f, N)` — suavizado por media en una ventana $N \times N$, análoga al *avg-pooling*, también sin reducción de resolución.
- `mm.readImg(h, w, dtype='float')` — útil para leer el mapa de entrada en punto flotante.

9.10.2.5 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Enteros H y W.
- Siguientes H líneas: W valores reales cada una.
- Siguiente línea: Enteros k y s.
- Siguiente línea: `max` o `avg`.

Salida:

- Línea 1: Enteros O_h y O_w.
- Siguientes O_h líneas: O_w valores reales cada una, con 4 decimales.

9.10.2.6 📌 Ejemplos

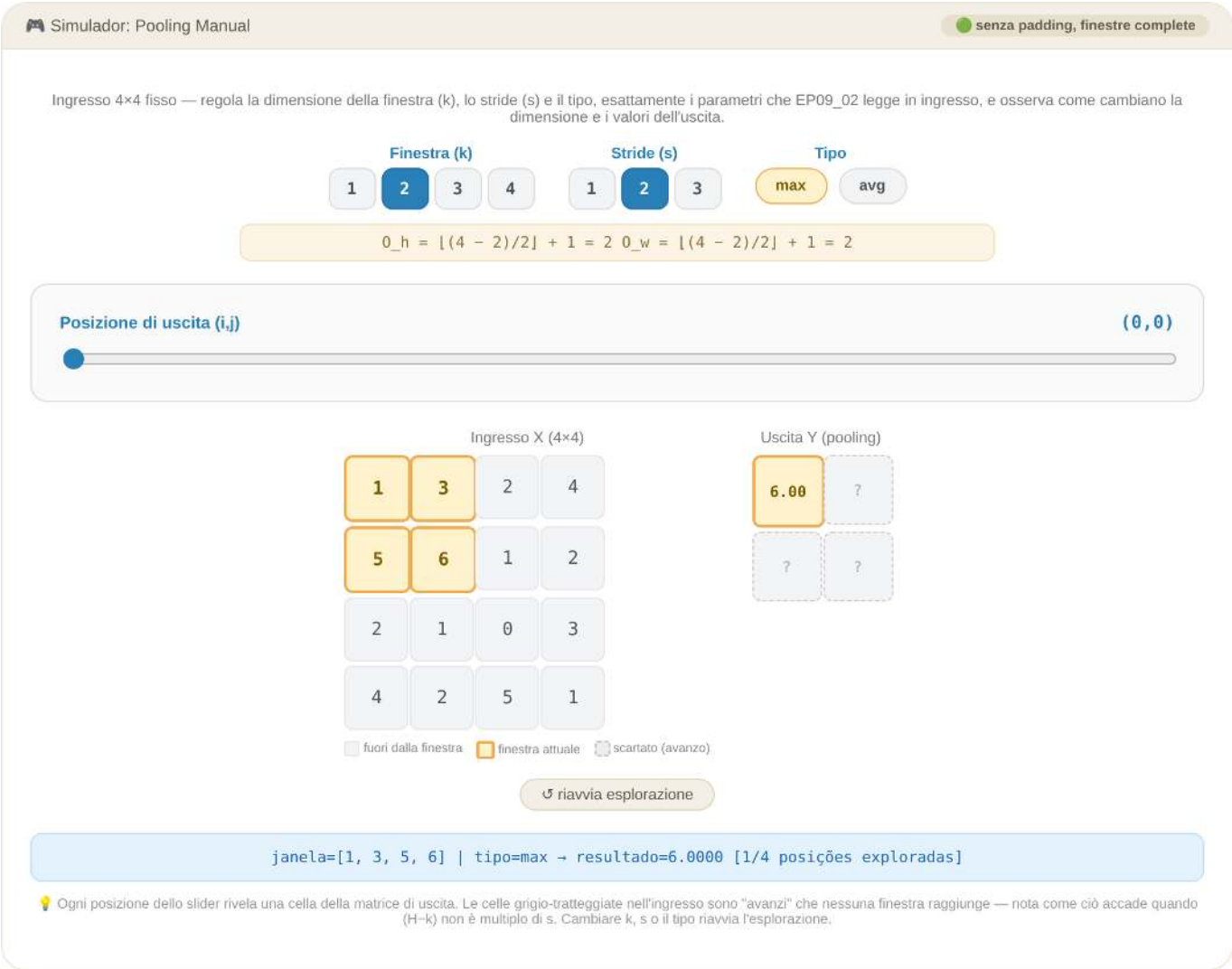
Entrada	Salida	Observación
4 4 1 3 2 4 5 6 1 2 2 1 0 3 4 2 5 1 2 2 max	2 2 6.0000 4.0000 4.0000 5.0000	*Pooling* máximo, ventana 2×2, *stride* 2.
4 4 1 3 2 4 5 6 1 2 2 1 0 3 4 2 5 1 2 2 avg	2 2 3.7500 2.2500 2.2500 2.2500	*Pooling* medio sobre las mismas ventanas.

```
%%writefile EP09_02.py
# Código Python
```

```
Overwriting EP09_02.py
```

```
TestSuite("EP09_02.py").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-ep0902.html

Figura 9.44: Simulador EP09_02: Pooling Manual (máximo vs. média, con ventana k y stride s ajustables)

9.10.3 EP09_03 🟡 Conteo de Parámetros Entrenables de una CNN

Este EP formaliza el conteo de parámetros entrenables de una *CNN*. Dada la descripción textual de una pequeña arquitectura, compuesta por capas convolucionales, de *pooling* y totalmente conectadas, determine, para cada capa, el número de parámetros entrenables y el total de la red.

La arquitectura debe interpretarse **secuencialmente**: la salida de una capa convolucional se convierte en la entrada de la siguiente capa compatible. Así, el número de canales producidos por una capa `CONV` determina el número de canales de entrada (`cin`) de la capa convolucional siguiente.

En una capa convolucional, es importante distinguir **canales de entrada** y **canales de salida**:

- c_{in} (*channels in*) es el número de **canales que entran en la capa**. Una imagen en tonos de gris posee $c_{in} = 1$, mientras que una imagen RGB posee $c_{in} = 3$. En una capa convolucional intermedia, `cin` normalmente es igual al número de canales producidos por la capa `CONV` anterior.
- c_{out} (*channels out*) es el número de **canales producidos por la capa**. Es igual al número de filtros utilizados. Por lo tanto, si una capa posee 16 filtros, produce $c_{out} = 16$ canales.

Por ejemplo, considere la secuencia:

```
CONV 3 3 1 8 1
POOL
CONV 3 3 8 16 1
POOL
FC 784 10 1
```

La primera convolución recibe una imagen con un canal y produce 8 canales. Después del *pooling*, la segunda convolución recibe esos 8 canales y produce 16 canales. La capa `POOL` no altera el número de canales, solo puede reducir las dimensiones espaciales. La capa `FC` recibe la cantidad de entradas informada en la propia descripción.

Cada filtro convolucional posee dimensiones

$$k_h \times k_w \times c_{in}.$$

Así, una capa con c_{out} filtros posee

$$k_h \cdot k_w \cdot c_{in} \cdot c_{out}$$

pesos. Si hay sesgo, se añade un parámetro para cada filtro, totalizando más c_{out} parámetros.

El punto central de este ejercicio es observar que la cantidad de parámetros de una capa convolucional **no depende de las dimensiones espaciales** ($H \times W$) del mapa de características. Esto ocurre debido al **comparticionamiento de pesos**: el mismo filtro se reutiliza en diferentes posiciones de la entrada.

9.10.3.1 📋 Directrices de Implementación

1. **Entrada:** Leer el entero L (número de capas de la arquitectura, en el orden en que se aplican).
2. **Capas:** Leer L líneas, cada una describiendo una capa en uno de los tres formatos:
   - `CONV kh kw cin cout bias` — capa convolucional con *kernel* $k_h \times k_w$, c_{in} canales de entrada, c_{out} canales de salida y `bias` (0 o 1), indicando si hay sesgo por filtro;
   - `POOL` — capa de *pooling* (máximo o medio), que no posee parámetros entrenables y preserva el número de canales;
   - `FC in out bias` — capa totalmente conectada con `in` entradas, `out` salidas y `bias` (0 o 1), indicando si hay sesgo por neurona.

3. **Consistencia entre capas `CONV`:** en una secuencia de capas convolucionales, el `cin` de una capa debe corresponder al `cout` de la capa convolucional anterior. Una capa `POOL` no altera ese número de canales.

   Por ejemplo:

   ```
   CONV 3 3 1 8 1
   POOL
   CONV 3 3 8 16 1
   ```

   La primera `CONV` produce 8 canales, que son recibidos por la segunda `CONV`. Por lo tanto, en la segunda capa, `cin=8` y `cout=16`.

4. **Parámetros de una capa `CONV`:**

   Cada uno de los c_{out} filtros posee $k_h \cdot k_w \cdot c_{in}$ pesos. Por lo tanto,

   $$P_{\text{CONV}} = k_h \cdot k_w \cdot c_{in} \cdot c_{out} + c_{out} \cdot \text{bias}.$$

5. **Parámetros de una capa `FC`:**

   $$P_{\text{FC}} = \text{in} \cdot \text{out} + \text{out} \cdot \text{bias}.$$

6. **Parámetros de una capa `POOL`:** siempre 0.
7. **Total de la red:** sumar los parámetros entrenables de todas las capas.
8. **Salida:** Para cada capa, en el orden de lectura, imprimir `Camada i: P`, donde i comienza en 1 y P es el número de parámetros de esa capa. Al final, imprimir `Total: T`.

9.10.3.2 📐 Ejemplo para entender `cin` y `cout`

Considere la secuencia:

```
CONV 3 3 1 8 1
POOL
CONV 3 3 8 16 1
```

En la primera capa:

- `cin=1`: entra un canal;
- `cout=8`: existen 8 filtros y, por lo tanto, salen 8 canales.

Cada filtro posee

$$3 \cdot 3 \cdot 1 = 9$$

pesos. Como existen 8 filtros:

$$9 \cdot 8 = 72$$

pesos. Con un sesgo por filtro:

$$72 + 8 = 80.$$

En la segunda capa:

- `cin=8`: entran los 8 canales producidos por la primera `CONV`;

- `cout=16`: existen 16 filtros y, por lo tanto, salen 16 canales.

Cada filtro posee

$$3 \cdot 3 \cdot 8 = 72$$

pesos. Como existen 16 filtros:

$$72 \cdot 16 = 1152$$

pesos. Con 16 sesgos:

$$1152 + 16 = 1168.$$

Así, las dos capas poseen, respectivamente, **80** y **1168 parámetros entrenables**.

Observe que `cout` **no es** *cin* multiplicado por el número de filtros. El número de filtros es exactamente `cout`: cada filtro combina todos los canales de entrada y produce **un único canal de salida**.

9.10.3.3 📌 Restricciones Computacionales

- **Independencia de la dimensión espacial:** la entrada no informa $H \times W$. El conteo de una capa `CONV` depende solo de `kh`, `kw`, `cin` y `cout`.
- **Consistencia de los canales:** para dos capas `CONV` consecutivas, el `cin` de la segunda debe ser igual al `cout` de la primera. Una capa `POOL` preserva el número de canales.
- **`bias` siempre 0 o 1:** multiplique directamente el término de sesgo por ese valor.
- **Capas `POOL` sin argumentos adicionales:** la línea contiene solo la palabra `POOL`.
- **Capas `FC`:** el número de entradas `in` se proporciona explícitamente. No es necesario calcular las dimensiones espaciales producidas por las capas anteriores.
- Todos los valores numéricos de entrada son enteros no negativos.

9.10.3.4 🧠 Fundamentación Teórica

Elemento	Papel en el conteo de parámetros
c_{in}	Número de canales recibidos por la capa
c_{out}	Número de filtros y, por lo tanto, de canales producidos por la capa
Filtro convolucional	Cada filtro posee $k_h \cdot k_w \cdot c_{in}$ pesos y produce un canal de salida
Compartición de pesos	El mismo filtro se reutiliza en diferentes posiciones de la entrada, haciendo el conteo independiente de $H \times W$
Sesgo	Un único parámetro adicional por filtro (`CONV`) o por neurona (`FC`)
Pooling	Puede alterar $H \times W$, pero no posee parámetros entrenables y preserva el número de canales
Capa `FC`	Posee un peso para cada combinación entre entrada y neurona de salida

9.10.3.5 🧩 Métodos del `morph.py` que pueden ayudar

Este ejercicio es puramente aritmético y no utiliza directamente funciones del `morph.py`. El conteo puede, sin embargo, verificarse en una arquitectura real implementada en PyTorch mediante:

```
sum(p.numel() for p in modelo.parameters())
```

Esta expresión contabiliza los parámetros del modelo, incluyendo pesos y sesgos.

9.10.3.6 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Entero L.
- Próximas L líneas: descripción de cada capa, en el formato `CONV kh kw cin cout bias`, `POOL` o `FC in out bias`.

Salida:

- L líneas en el formato `Camada i: P`.
- Última línea: `Total: T`.

9.10.3.7 📌 Ejemplos

Entrada	Salida	Observación
3 CONV 3 3 1 8 1 POOL FC 1352 10 1	Camada 1: 80 Camada 2: 0 Camada 3: 13530 Total: 13610	Red simple con una convolución, *pooling* y capa de clasificación.
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	Pequeña CNN con dos convoluciones, dos *poolings* y una capa totalmente conectada.
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	CNN pequeña con dos convoluciones, *pooling* intermedio y dos capas totalmente conectadas para clasificación.

```
%%writefile EP09_03.py
# Código Python
```

```
Overwriting EP09_03.py
```

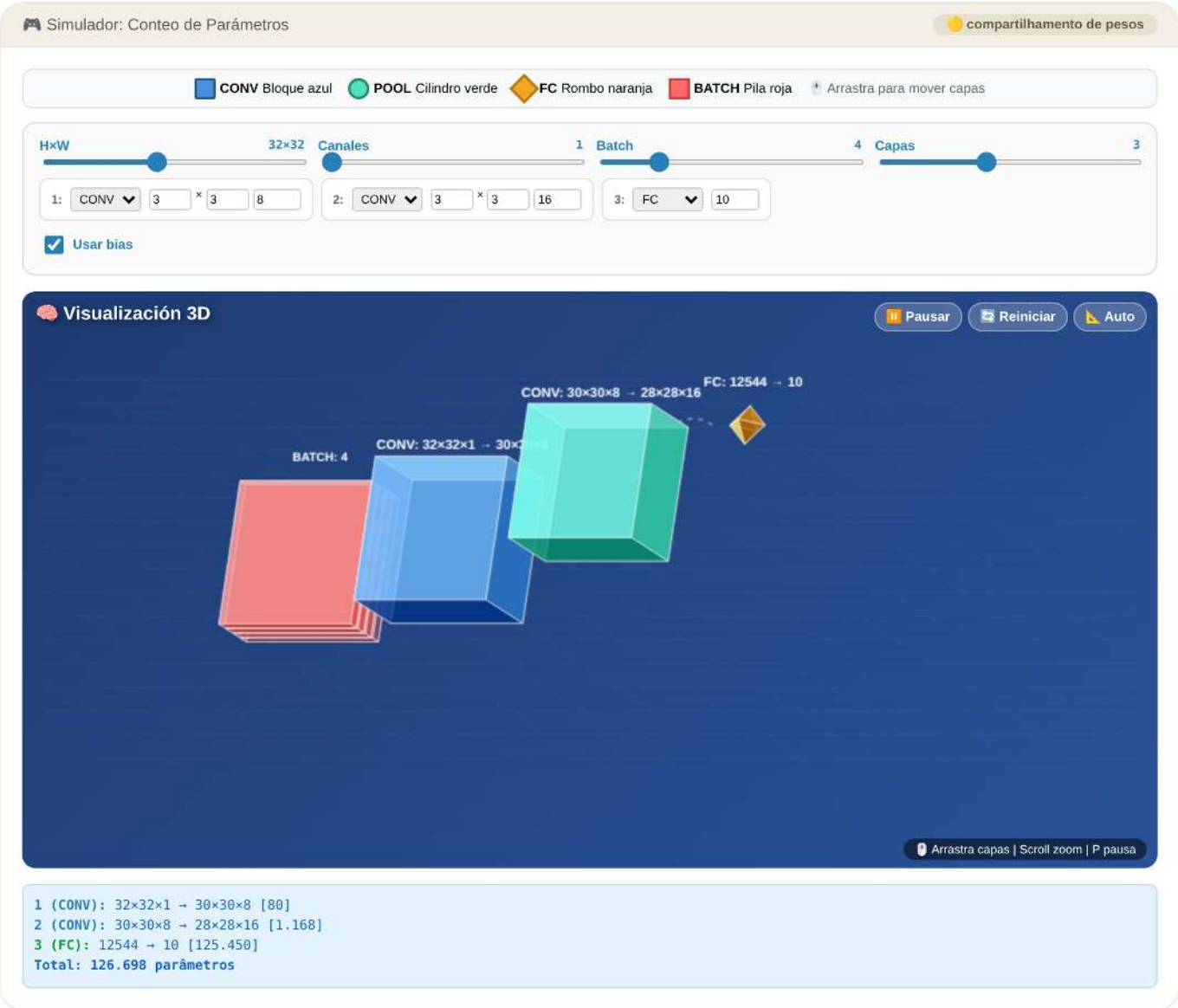
```
TestSuite("EP09_03.py").run()
```

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

9.10.4 EP09_04 🟡 Intersección sobre Unión (IoU) y Supresión de No-Máximos (NMS)

Los modelos de detección de objetos pueden producir **varias cajas delimitadoras candidatas** para un mismo objeto, con diferentes posiciones y puntuaciones de confianza. La etapa de posprocesamiento responsable de eliminar esas detecciones redundantes es la **Supresión de No-Máximos (NMS)**, cuya operación fundamental utiliza la métrica de **Intersección sobre Unión (IoU)**.

La NMS utiliza esta medida para decidir qué cajas deben mantenerse. En general, la caja con mayor confianza se selecciona primero; a continuación, las cajas que presentan una IoU por encima de un determinado umbral con la caja seleccionada se consideran redundantes y se eliminan. El proceso se repite hasta que no queden cajas candidatas.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-ep0903.html

Figura 9.45: Simulador EP09_03: Conteo de Parámetros — Convolución vs. Capa Totalmente Conectada

En este ejercicio, deberás implementar el algoritmo de NMS desde cero, calculando la IoU entre cajas y aplicando sucesivamente el criterio de selección y supresión para producir el conjunto final de detecciones.

9.10.4.1 📋 Directrices de Implementación

1. **Entrada:** Leer el entero N (número de cajas candidatas) y el umbral real τ (umbral de IoU para la supresión), en la misma línea.
2. **Cajas:** Leer N líneas, cada una con cinco valores reales:

   ```
   x1 y1 x2 y2 score
   ```

   donde (x_1, y_1) representa la esquina superior izquierda, (x_2, y_2) la esquina inferior derecha y `score` la puntuación de confianza.
3. **Intersección sobre Unión:** Para dos cajas A y B,

   $$IoU(A, B) = \frac{\text{Área}(A \cap B)}{\text{Área}(A \cup B)}.$$

   El área de intersección debe calcularse a partir de la superposición de los intervalos en x e y. Si no hay superposición, el área de intersección es cero.
4. **Algoritmo voraz de NMS:**
   a. Ordena las cajas por `score` descendente. En caso de empate, mantén el orden original de lectura.
   b. Selecciona la caja de mayor puntuación entre las cajas restantes y añádela al conjunto de salida.
   c. Calcula la IoU entre la caja seleccionada y **todas las cajas aún restantes**. Suprime las cajas para las cuales

   $$\text{IoU} > \tau.$$

   d. Repite los pasos (b) y (c) hasta que no queden cajas.
5. **Salida:** Para cada caja mantenida, en el orden en que fue seleccionada, imprimir su índice original (posición de lectura, comenzando en 0) y su `score`, formateado con 4 decimales. Al final, imprimir:

   ```
   Total mantenidas: X
   ```

9.10.4.2 📌 Restricciones Computacionales

- **Supresión estricta:** solo las cajas con IoU $> \tau$ se suprimen. Las cajas con IoU $= \tau$ se mantienen.
- **Índices originales:** la salida hace referencia a la posición en que cada caja fue leída en la entrada (comenzando en 0), y no a su posición después de la ordenación.
- **Ordenación estable:** en caso de `score` iguales, debe preservarse el orden original de lectura.
- **Rectángulos alineados a los ejes:** todas las cajas se especifican mediante dos esquinas, con $x_1 < x_2$ e $y_1 < y_2$ garantizados en la entrada.
- **Coordenadas y puntuaciones:** los valores reales pueden ser positivos o negativos, según los límites definidos por la entrada, pero las dimensiones de las cajas son siempre positivas.

9.10.4.3 🧠 Fundamentación Teórica

Elemento	Papel en el posprocesamiento de detección
IoU	Cuantifica la superposición espacial entre dos cajas; IoU $= 1$ para cajas idénticas y IoU $= 0$ para cajas sin superposición

Elemento	Papel en el posprocesamiento de detección
Ordenación por confianza	Hace que la caja de mayor `score` se analice primero
Umbral τ	Define la cantidad de superposición necesaria para que una caja se considere redundante
Supresión	Elimina cajas que presentan una gran superposición con una caja ya seleccionada
Cajas distantes	Poseen IoU cercana a cero y, en general, no se suprimen por esta regla

9.10.4.4 🧩 Métodos de `morph.py` que pueden ayudar

- `mm.IoU(boxA, boxB)` — calcula la métrica de IoU, pero espera las cajas en el formato (x, y, w, h), es decir, esquina superior izquierda, ancho y alto. La entrada de este ejercicio utiliza el formato (x_1, y_1, x_2, y_2). La conversión es directa:

$$w = x_2 - x_1, \qquad h = y_2 - y_1.$$

  El uso de esta función es opcional. El objetivo principal del ejercicio es implementar correctamente el proceso de selección y supresión de la NMS.

9.10.4.5 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: entero N y real τ.
- Siguientes N líneas: x_1 y_1 x_2 y_2 score.

Salida:

- Una línea por caja mantenida, en el orden de selección: `índice score`.
- Última línea: `Total mantenidas: X`.

```
%%writefile EP09_04.py
# Código Python
```

```
Overwriting EP09_04.py
```

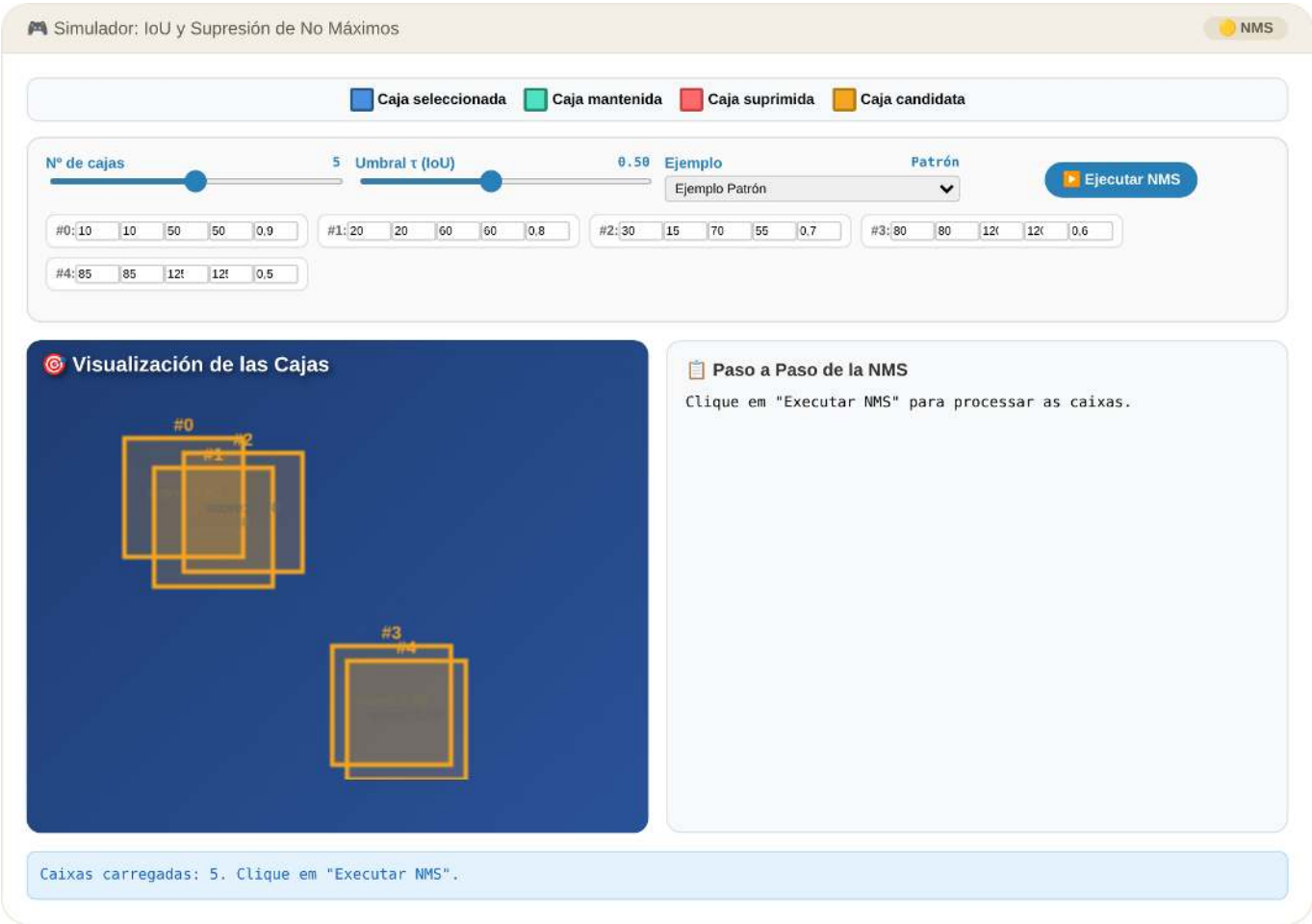
```
TestSuite("EP09_04.py").run()
```

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

9.10.5 EP09_05 🟠 Evaluación de Segmentación: IoU y Dice Pixel a Pixel

El Bloque 2 de la sección "Segmentación Semántica con Arquitectura U-Net" define, en pocas líneas, la función `iou_mascaras`, utilizada para medir la calidad de la línea base morfológica clásica (suavizado + Otsu + apertura) y, más adelante, de la propia U-Net entrenada. A diferencia del IoU del EP09_04 — calculado sobre **cajas delimitadoras** (regiones rectangulares descritas por cuatro números) —, el IoU de segmentación se calcula **pixel a pixel**: cada posición de la imagen se compara individualmente entre la máscara predicha y la máscara de referencia.

Se le ha encargado generalizar esta evaluación, implementando no solo el IoU pixel a pixel, sino también el **coeficiente de Dice**, otra métrica de superposición ampliamente utilizada en segmentación médica (incluso en la función `perda_dice`, mencionada en el mismo bloque del capítulo como base de la función de pérdida utilizada para entrenar la U-Net).



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-ep0904.html

Figura 9.46: Simulador EP09_04: IoU y Supresión de No-Máximos (NMS)

9.10.5.1 📋 Directrices de Implementación

1. **Entrada:** Leer las dimensiones $H \times W$ de las máscaras.
2. **Máscara predicha:** Leer H líneas con W valores enteros (0 o 1) cada una — por ejemplo, la salida de una U-Net después de la umbralización en 0,5 sobre la sigmoide, como en el Bloque 4 del capítulo.
3. **Máscara de referencia:** Leer otras H líneas con W valores enteros (0 o 1) cada una — el *ground truth*.
4. **Intersección y unión:** Considerando cada píxel como perteneciente al objeto cuando su valor es diferente de cero,
$$\text{intersección} = \sum_{i,j} \mathbb{1}[P_{ij} = 1 \wedge R_{ij} = 1], \qquad \text{unión} = \sum_{i,j} \mathbb{1}[P_{ij} = 1 \vee R_{ij} = 1].$$
5. **IoU pixel a pixel:**
$$\text{IoU} = \frac{\text{intersección}}{\text{unión}}.$$
6. **Coeficiente de Dice:**
$$\text{Dice} = \frac{2 \cdot \text{intersección}}{|P| + |R|},$$
donde $|P|$ y $|R|$ son el número total de píxeles de objeto en cada máscara.
7. **Convención para máscaras vacías:** si **ambas** máscaras no poseen ningún píxel de objeto (unión $= 0$ y $|P| + |R| = 0$), considere la correspondencia trivialmente perfecta: IoU = Dice = 1,0.
8. **Salida:** Dos líneas, `IoU: X.XXXX` y `Dice: X.XXXX`, cada valor con 4 decimales.

9.10.5.2 📌 Restricciones Computacionales

- **Cualquier valor no nulo cuenta como objeto:** trate valores diferentes de 0 (no solo 1) como pertenecientes a la máscara, replicando la comprobación `predita > 0` utilizada en `iou_mascaras` en el capítulo.
- **Mismas dimensiones:** las dos máscaras siempre poseen exactamente $H \times W$ elementos.
- **Convención de vacío:** aplique la regla del ítem 7 **solo** cuando ambas máscaras estén totalmente vacías; si solo una está vacía, la intersección es 0 y el IoU/Dice resultante también será 0.

9.10.5.3 🧠 Fundamentación Teórica

Elemento	Papel en la evaluación de segmentación
IoU pixel a pixel	Generaliza la métrica del EP09_04 para regiones de forma arbitraria — no solo rectángulos — comparando máscara predicha y referencia posición a posición
Coeficiente de Dice	Métrica relacionada con el IoU (siempre Dice $\geq$ IoU), más sensible a pequeñas intersecciones y ampliamente utilizada como función de pérdida en segmentación (función `perda_dice` del capítulo)
Convención de máscaras vacías	Evita la división por cero y reconoce que "ningún objeto previsto, ningún objeto real" es, por definición, un acierto
Comparación clásico vs. U-Net	El capítulo usa exactamente este tipo de métrica para justificar, numéricamente, por qué la U-Net supera la línea base morfológica en escenarios de bajo contraste

9.10.5.4 🧩 Métodos del `morph.py` que pueden ayudar

- `mm.readImg(h, w, dtype='uint8')` — lee directamente cada máscara binaria $h \times w$ de la entrada estándar (los valores 0/1 caben perfectamente en el tipo entero estándar).
- La propia función `iou_mascaras`, definida en el Bloque 2 de la sección de U-Net del capítulo (no forma parte del `morph.py`, sino del código del capítulo), es la inspiración directa de este ejercicio — vale la pena releer esas pocas líneas antes de programar.
- Para una extensión opcional (no exigida por este EP), `mm.connectedComponents` o `mm.label0` (vistos en el contexto de análisis de componentes conexos) permitirían etiquetar cada nódulo individualmente y calcular el IoU **por componente**, en lugar de sobre la máscara completa.

9.10.5.5 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Enteros H y W.
- Siguientes H líneas: W valores enteros (0 o 1) — máscara predicha.
- Siguientes H líneas: W valores enteros (0 o 1) — máscara de referencia.

Salida:

- Línea 1: `IoU: X.XXXX`.
- Línea 2: `Dice: X.XXXX`.

💡 Ejemplo Ilustrativo

Considere una máscara predicha con un cuadrado 2×2 de píxeles activos y una referencia desplazada en una columna, superponiéndose en solo la mitad del área:

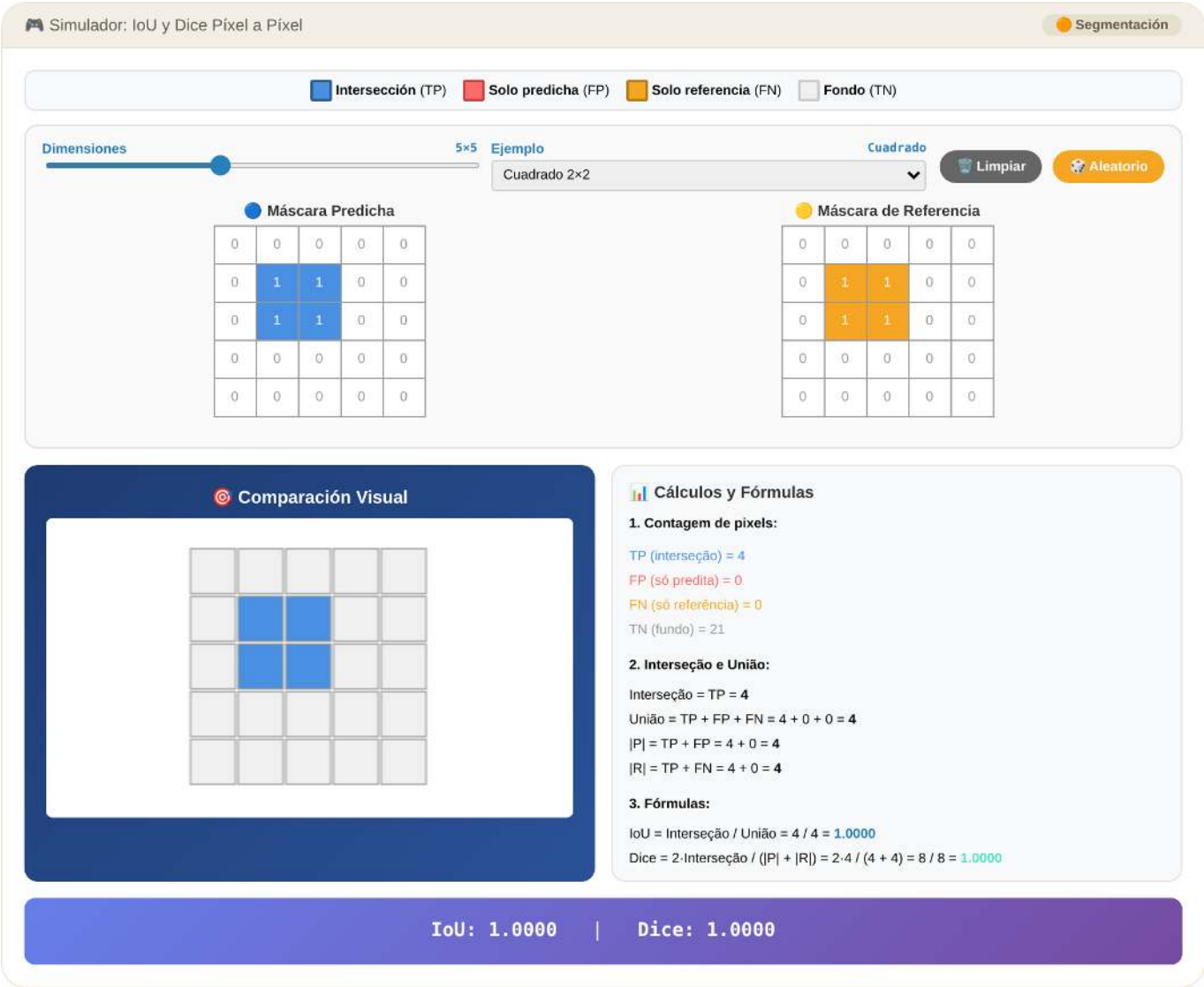
```
Predicha          Referencia
0 0 0 0          0 0 0 0
0 1 1 0          0 0 1 1
0 1 1 0          0 0 1 1
0 0 0 0          0 0 0 0
```

Intersección $= 2$ píxeles, unión $= 6$ píxeles $(4 + 4 - 2)$, por lo tanto IoU $= 2/6 \approx 0{,}3333$ y Dice $= 2 \cdot 2/(4 + 4) = 0{,}5000$ — observe que el Dice es siempre igual o mayor que el IoU para la misma superposición.

9.10.5.6 📌 Ejemplos

Entrada	Salida	Observación
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	Máscaras 4×4 con superposición parcial de 2 píxeles.

```
%%writefile EP09_05.py
# Código Python en español
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-ep0905.html

Figura 9.47: Simulador EP09_05: Evaluación de Segmentación — IoU y Dice Píxel a Píxel

```
Overwriting EP09_05.py
```

```
TestSuite("EP09_05.py").run()
```

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

9.10.6 EP09_06 🔴 *Pipeline* Integrado: De la Detección a la Medición del Mundo Real

Este ejercicio final integra los dos ejercicios de detección y el principio de **fotogrametría** presentado en la sección "Fotogrametría y Referencia de Escala" — exactamente el mismo cálculo implementado en la figura de medición por referencia de escala de este capítulo. El escenario reproduce una situación realista: un detector (Faster R-CNN o YOLO) genera **varias cajas candidatas superpuestas** para el mismo objeto de interés; tras filtrarlas por NMS, la caja superviviente de mayor confianza se usa, junto con una caja de referencia de ancho real conocido (como la tarjeta de 8,56 cm), para estimar las dimensiones reales del objeto detectado.

9.10.6.1 📋 Directrices de Implementación

1. **Referencia conocida:** Leer el valor real L_{ref} (ancho real del objeto de referencia, en cm) y, a continuación, los cuatro reales x_1 y_1 x_2 y_2 de su caja delimitadora en píxeles (ya conocida, sin necesidad de detección).
2. **Candidatas del objeto a medir:** Leer el entero N (número de cajas candidatas producidas por el detector para el objeto de interés) y el umbral real τ; a continuación, leer las N líneas de cajas candidatas, cada una con x_1 y_1 x_2 y_2 score.
3. **Etapa 1 — NMS:** Aplicar exactamente el algoritmo de Supresión de No Máximos del EP09_04 a las N cajas candidatas, usando el umbral τ, para eliminar detecciones redundantes del mismo objeto.
4. **Etapa 2 — Selección de la caja final:** Tras el NMS, la caja de mayor `score` entre las mantenidas es la detección final del objeto (la entrada garantiza que todas las cajas candidatas corresponden a un único objeto físico, por lo que la primera caja seleccionada por el NMS ya es el resultado final).
5. **Etapa 3 — Medición por referencia de escala:** Calcular la razón cm/pixel = L_{ref}/ancho de la referencia en y aplicarla tanto al ancho como a la altura (en píxeles) de la caja final del objeto, obteniendo sus dimensiones reales estimadas en centímetros.
6. **Salida:** Primero, una línea por caja mantenida tras el NMS (mismo formato del EP09_04): `índice score`. A continuación, la línea `Total mantenidas: X`. Finalmente, la línea `Objeto: L x A cm`, donde L y A son el ancho y la altura estimados del objeto, cada uno con 2 decimales.

9.10.6.2 📌 Restricciones Computacionales

- **Reutilizar el NMS del EP09_04** íntegramente — misma regla de desempate, mismo criterio de supresión (IoU $> \tau$).
- **La referencia no pasa por NMS:** su caja se da directamente, sin candidatas competidoras.
- **Razón única para ancho y altura:** así como en la figura de fotogrametría del capítulo, la misma razón cm/pixel (derivada del ancho de la referencia) se aplica tanto al ancho como a la altura del objeto — no hay calibración vertical separada.

9.10.6.3 🧠 Fundamentación Teórica

Etapa	Concepto del capítulo
Múltiples cajas candidatas	Salida bruta de un detector como el Faster R-CNN o el YOLO, antes del posprocesamiento
NMS (EP09_04)	Filtra las detecciones redundantes, preservando solo la más confiable para el objeto

Etapa	Concepto del capítulo
Referencia de escala conocida	Mismo principio de la tarjeta de 8,56 cm usado en la sección "Fotogrametría y Referencia de Escala"
Conversión píxel → centímetro	Regla de tres simple: cm/pixel = L_{ref}/w_{ref_px}, aplicada a la caja final del objeto

9.10.6.4 🧩 Métodos del `morph.py` que pueden ayudar

- `mm.IoU(boxA, boxB)` — la misma función sugerida en el EP09_04, aquí reutilizada dentro de la etapa de NMS de este *pipeline* integrado (recuerde la conversión de formato: $w = x_2 - x_1$, $h = y_2 - y_1$).
- Si ya resolvió el EP09_04 encapsulando el NMS en una función propia, este es el momento ideal de **reutilizar ese código** — la integración de módulos ya probados individualmente es exactamente la práctica de ingeniería que este ejercicio quiere reforzar.

9.10.6.5 📦 Especificación de Entrada y Salida (VPL)

Entrada:

- Línea 1: Real L_{ref}.
- Línea 2: x_1 y_1 x_2 y_2 de la caja de referencia.
- Línea 3: Entero N y real τ.
- Siguientes N líneas: x_1 y_1 x_2 y_2 score de las cajas candidatas del objeto.

Salida:

- Una línea por caja mantenida tras el NMS: `índice score`.
- Línea siguiente: `Total mantenidas: X`.
- Última línea: `Objeto: L x A cm`.

9.10.6.6 📌 Ejemplos

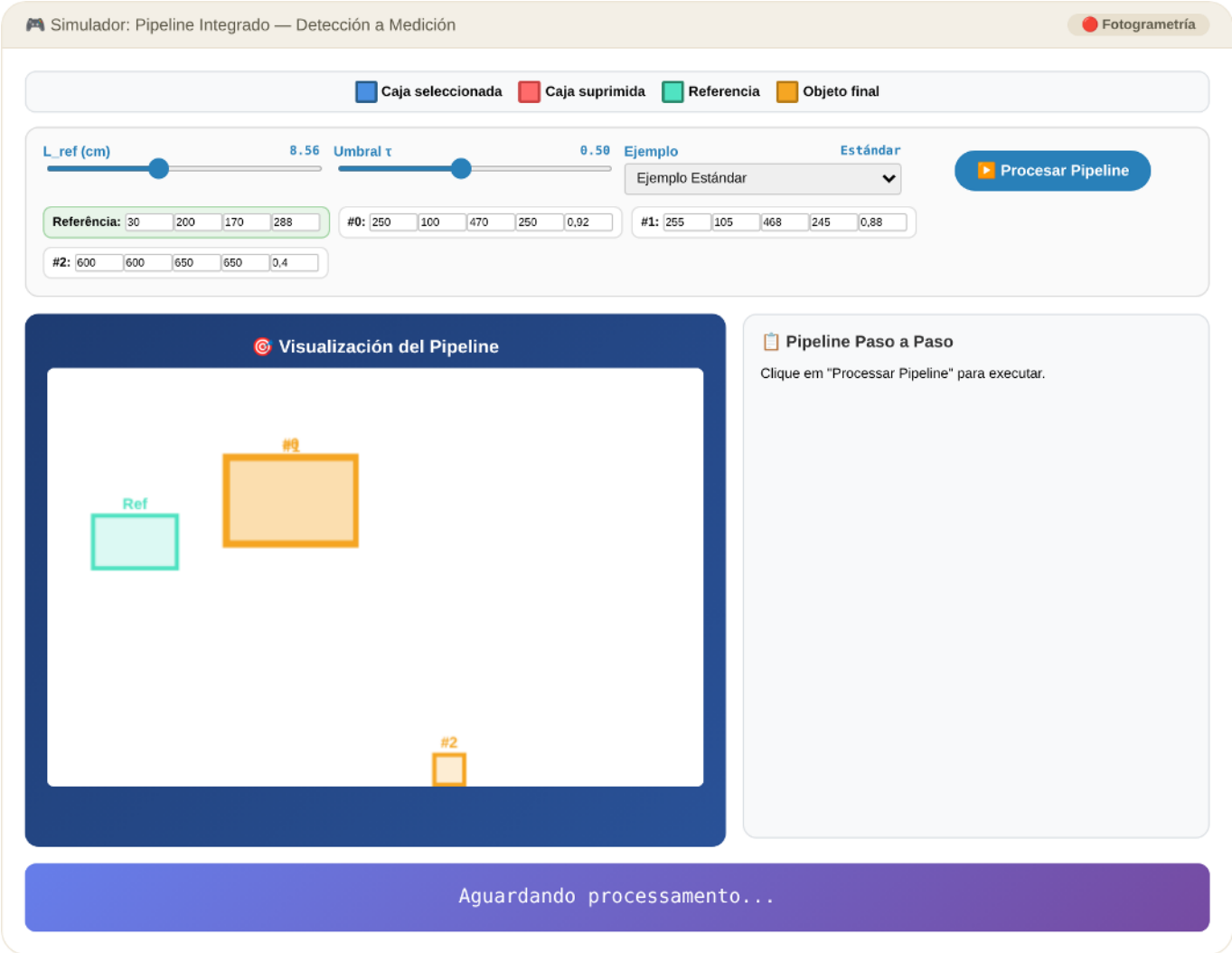
Entrada	Salida	Observación
8.56 30 200 170 288 3 0.5 250 100 470 250 0.92 255 105 468 245 0.88 600 600 650 650 0.40	0 0.9200 2 0.4000 Total mantenidas: 2 Objeto: 13.45 x 9.17 cm	La caja 1 se suprime por superponerse fuertemente a la caja 0; la detección final del objeto es la caja 0.

```
%%writefile EP09_06.py
# Código Python
```

```
Overwriting EP09_06.py
```

```
TestSuite("EP09_06.py").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.es/cap09/sim-ep0906.html

Figura 9.48: Simulador EP09_06: Pipeline Integrado — Detección a la Medición del Mundo Real

Referencias

BAHDANAU, Dzmitry; CHO, Kyunghyun; BENGIO, Yoshua. Neural Machine Translation by Jointly Learning to Align and Translate. *En*: 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. *En*: 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. *En*: 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. *En*: 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.

Apéndice A

Sobre o MCTest

O **MCTest** é um sistema de código aberto para criação e correção automática de exames parametrizados (Zampirolli, 2023). Ele oferece suporte a questões de múltipla escolha, dissertativas e de programação, estas últimas integradas ao VPL do Moodle, descrito no Apêndice B.

A versão do MCTest utilizada neste livro é a **5.4**, disponível em https://github.com/fzampirolli/mctest. A subpasta `book` do repositório contém as versões **5.3**, descrita em (Zampirolli, 2023), e **5.4**, utilizada para gerar as provas apresentadas neste livro e atualmente em produção na UFABC (https://mctest.ufabc.edu.br).

A.1 Instalação do MCTest

A instalação recomendada utiliza o VirtualBox com Ubuntu 22.04. No terminal do Ubuntu, como *root*, execute:

```
wget https://raw.githubusercontent.com/fzampirolli/mctest/master/_setup-all.sh
```

Em seguida, substitua `yourLogin` pelo nome do usuário local e execute:

```
sed -i 's/\/home\/fz\//\/home\/yourLogin\//g' _setup-all.sh
source _setup-all.sh
pip install mysqlclient
```

Após alguns minutos, o MCTest estará configurado. Para executá-lo:

```
source /home/yourLogin/PycharmProjects/runDjango.sh
```

O sistema fica então acessível em `http://127.0.0.1:8000`.

A.2 Criando um exame no MCTest

No MCTest, todo exame é configurado em uma tela de **Exame**, que reúne Turmas, Tópicos (associados a uma disciplina, como PDI-VC) e Questões — cada questão pertence a um único tópico. Além dos atributos de banco de dados, o exame possui parâmetros próprios, como o número de questões sorteadas e o número de variações geradas.

No caso das duas provas descritas neste apêndice, foi definido um conjunto de questões parametrizadas por prova, das quais um subconjunto é sorteado aleatoriamente para cada variação, totalizando **110 variações** distintas — uma por estudante matriculado nas turmas.

O botão **Create-Variations** gera um novo conjunto de variações a cada acionamento. Quando as questões estão associadas a atividades VPL, o professor recebe por e-mail um arquivo `*linker.json` contendo todos os casos de teste das variações geradas. O botão **Create-PDF** gera, para cada turma, um PDF com as provas sorteadas por estudante, além de um arquivo `*students_variations.csv` com nome, e-mail e variação sorteada de cada estudante.

! Atenção

Cada acionamento do botão **Create-Variations** gera um novo conjunto de variações de exames. Após imprimir o PDF, é fundamental **não alterar os atributos do exame**, sob risco de invalidar a correção automática das provas já impressas ou aplicadas.

A.3 Criando uma questão paramétrica

Uma questão paramétrica no MCTest combina três elementos:

1. **Descrição** — o enunciado da questão, escrito em LaTeX, contendo variáveis entre `[[code:variavel]]`, que são substituídas pelo valor sorteado para cada estudante;
2. **Bloco de definição** (`[[def: ... ]]`) — um trecho de código Python, embutido na própria questão, responsável por sortear os parâmetros, calcular a solução de referência e montar os casos de teste;
3. **Casos de teste para o Moodle** (bloco `moodle_cases`) — dicionário serializado em JSON contendo as entradas e saídas esperadas, consumido pela atividade VPL.

Um exemplo simplificado de definição de questão (adaptado da geração de um tabuleiro de xadrez `h × w`) é:

```
[[code:texto]]

\vspace{2mm}\noindent\textbf{Exemplo de Entrada:}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_inp]]
\end{verbatim}
\vspace{-2mm}\noindent\textbf{Exemplo de Saída:}\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 # TEM QUE INCLUIR "topic." no 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

# PARÂMETROS USADOS NA DESCRIÇÃO DA QUESTÃO
height = int(np.random.randint(5, 15))

# ENUNCIADO COMPLETO, COM O VALOR DE height JÁ INTERPOLADO
texto = (
    r"\vspace{2mm}\noindent\textbf{Descrição:}\vspace{-2mm} "
    r"Escreva um programa que leia um número inteiro \texttt{W}, representando a "
    r"largura (número de colunas) de um tabuleiro, e imprima esse tabuleiro no "
    r"formato de um tabuleiro de xadrez, usando os dígitos \texttt{0} e \texttt{1}. "
    r"O tabuleiro impresso terá sempre altura (número de linhas) igual a "
    f"\\textbf{{{height}}} "
    r"e largura igual ao valor \texttt{W} lido da entrada. "
```

```
    r"Considerando a posição $(i, j)$ do tabuleiro (indexada a partir de 0, com "
    r"$i$ representando a linha e $j$ a coluna), o valor impresso nessa posição "
    r"deve ser \texttt{1} se $i + j$ for ímpar, e \texttt{0} caso contrário. Cada "
    r"linha do tabuleiro deve ser impressa em uma linha separada, com os valores "
    r"de cada coluna separados por um espaço."
)

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']      = ["matrizes", "laços de repetição", "formatação de saída"]
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]
]]
```

O valor de `height` é sorteado uma única vez por variação (para aparecer no enunciado), enquanto `width` é sorteado a cada caso de teste, aumentando a robustez da correção automática.

Vale destacar o papel duplo da variável `texto` nesse mecanismo. Ela é ao mesmo tempo:

- **conteúdo do enunciado impresso**, inserida na descrição LaTeX da questão através da tag `[[code:texto]]`, aparecendo no PDF gerado pelo botão **Create-PDF**; e
- **entrada do dicionário exportado para o Moodle**, através da chave `cases['description']`, que passa a compor o JSON `moodle_cases`. É justamente esse campo que a atividade VPL exibe ao estudante quando ele abre a questão no Moodle para avaliá-la ou submeter uma solução.

Há, porém, uma diferença importante entre esses dois usos: o PDF é composto em LaTeX e, portanto, interpreta normalmente comandos como `\textbf{}`, `\texttt{}` ou `$...$`. Já o VPL do Moodle **não** processa LaTeX — ele espera texto puro. Por isso, ao montar `cases['description']`, `texto` não é inserido diretamente, mas sim passado pela função utilitária `latex_to_text()`, própria do MCTest, que remove (ou converte) os comandos e símbolos LaTeX do enunciado, produzindo uma versão em texto simples equivalente. Assim, `[[code:texto]]` no PDF continua recebendo o `texto` original, com toda a formatação LaTeX, enquanto `cases['description']` recebe `latex_to_text(texto)`, garantindo que o enunciado exibido no Moodle seja legível mesmo sem suporte a LaTeX.

Como `texto` é montada dentro do próprio bloco `[[def: ...]]` — no mesmo escopo em que `height`, `width` e os demais parâmetros são sorteados —, ela é recalculada a cada variação. Isso garante que o enunciado visto pelo estudante no Moodle seja **sempre idêntico** ao que foi impresso na prova em papel, mesmo quando o texto muda de estudante para estudante junto com os valores sorteados. A Seção A.4 retoma esse ponto em um caso real, no qual o enunciado é significativamente mais longo e descreve, além dos parâmetros numéricos, o próprio cenário visual gerado para cada variação.

A Figura A.1 mostra a questão do tabuleiro de xadrez efetivamente gerada pelo MCTest para uma das variações, com o enunciado já contendo o valor sorteado de `height` e o exemplo de entrada/saída correspondente ao primeiro caso de teste:

Ao acionar **Create-PDF** na tela da questão, uma nova variação é gerada a cada clique, permitindo conferir o enunciado antes de publicá-lo.

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
```

Figura A.1: Questão do tabuleiro de xadrez gerada pelo MCTest, ilustrando o enunciado parametrizado com o valor de `height` sorteado para a variação e o exemplo de entrada/saída do primeiro caso de teste.

Para as duas provas da disciplina, esse mecanismo foi utilizado de forma mais ampla: cada questão paramétrica define não apenas os valores numéricos, mas também, em alguns casos, pequenas variações estruturais no enunciado (por exemplo, qual direção cardinal — Norte, Sul, Leste, Oeste — deve ser avaliada), dificultando a busca por soluções prontas na internet ou em ferramentas de IA generativa.

A.4 Exemplo real: uma questão do Simulado 4

Para ilustrar o mecanismo descrito na seção anterior com um caso concreto, apresenta-se a seguir uma questão efetivamente aplicada no **Simulado 4** (tópico *im4-Operadores Morfológicos*, dificuldade 1, taxonomia de Bloom "remember"), do tipo questão de programação parametrizada com integração Moodle+VPL.

A questão pede ao estudante que escreva um programa capaz de:

- ler uma imagem binária, corrompida por ruído sal e pimenta, contendo ao menos dois objetos geométricos isolados;
- aplicar filtragem morfológica (abertura seguida de fechamento) para eliminar o ruído;
- extrair, a partir da imagem filtrada, as medidas geométricas de cada objeto (área, perímetro, centro, bounding box, circularidade, solidez e número de vértices), usando a função auxiliar `mm.measure`;
- ordenar os objetos por posição (coordenada X do bounding box, com Y como critério de desempate) e reatribuir os identificadores sequencialmente;
- imprimir a matriz limpa e a tabela de medidas no formato esperado pelo corretor automático.

No bloco `[[def: ... ]]` correspondente, um gerador de cena (`gerarCena`) sorteia aleatoriamente a altura e a largura da imagem, o tipo, o tamanho e a posição de cada objeto (quadrado, retângulo, triângulo ou bloco), garantindo que eles não se sobreponham, e em seguida acrescenta ruído sal e pimenta com 3% de probabilidade por pixel. Dez casos de teste distintos são gerados dessa forma para cada variação da prova, e o par entrada/saída do primeiro caso é reaproveitado no próprio enunciado como exemplo para o estudante. A biblioteca de morfologia (`mm`) é importada diretamente de `morph.py`, também disponível como módulo utilitário do próprio MCTest.

Em resumo, o enunciado pede ao estudante um programa que:

1. leia, na primeira linha, dois inteiros H e W (altura e largura da imagem);
2. leia as H linhas seguintes da matriz binária (0s e 1s separados por espaço), usando `mm.readImg(H, W)`;
3. aplique filtragem morfológica para remover o ruído sal e pimenta;
4. imprima a matriz já filtrada, em 0s e 1s separados por espaço;
5. extraia as medidas geométricas dos objetos com `mm.measure(imgLimpa)`;
6. ordene os objetos e imprima a tabela de medidas no formato esperado.

O enunciado ainda traz duas observações importantes para a correção automática: (i) a área calculada

pelo OpenCV (`cv2.contourArea`) corresponde ao polígono contínuo delimitado pelos centros dos pixels de borda, sendo por isso menor do que a simples contagem de pixels 1 (`np.sum`); e (ii) a lista de objetos deve ser ordenada em ordem crescente pela coordenada X do bounding box (`bbox[0]`), usando a coordenada Y (`bbox[1]`) como critério de desempate, com os IDs reatribuídos sequencialmente de 1 a N após a ordenação.

Assim como no exemplo da Seção A.3, o texto do enunciado aqui também não é fixo: ele é montado dentro do próprio bloco `[[def: ... ]]`, na variável Python `texto`, e desempenha o mesmo papel duplo já descrito — alimenta o PDF via `[[code:texto]]` e é exportado, já convertido por `latex_to_text()`, em `cases['description']` dentro de `moodle_cases`, sendo essa a versão exibida ao estudante no VPL do Moodle. A diferença é que, neste caso real, é o **cenário visual** (imagem ruidosa, número e posição dos objetos) que muda de estudante para estudante a cada variação, enquanto a redação de `texto` permanece fixa entre variações, já que apenas os dados numéricos (imagem e casos de teste) são sorteados. O ideal seria que a redação também variasse a cada geração, como no exemplo anterior, em que o valor de `height` era interpolado diretamente no texto da descrição. A estrutura real utilizada (com trechos de sorteio de cenário omitidos por brevidade) é:

```
[[code:texto]]

\vspace{2mm}\noindent\textbf{Exemplo de Entrada:}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_inp]]
\end{verbatim}
\vspace{-2mm}\noindent\textbf{Exemplo de Saída:}\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 # TEM QUE INCLUIR "topic." no MCTest

# ... geração da cena, do ruído e dos 10 casos de teste (inplist/outlist) ...

# ENUNCIADO COMPLETO, COM EXPLICAÇÃO DE ÁREA E DE ORDENAÇÃO
texto = (
    "Uma imagem binária corrompida por ruído sal e pimenta contém no mínimo dois objetos
    geométricos isolados.\n\n"
    "Escreva um programa que:\n"
    "1. Leia dois inteiros ..."
)

cases = {}
cases['skills']      = ["morfologia matematica", "remocao de ruido", "mm.measure",
"tabulacao"]
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]
]]
```

Abaixo, a fotografia da questão efetivamente gerada e impressa para uma das 110 variações do Simulado

4, mostrando o enunciado, a imagem de entrada ruidosa e o exemplo de saída esperado (imagem filtrada e tabela de medidas):

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 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 1 1 1 1 1 1 0 1 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 1 1 1 1 1 1 1 1 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 0 1 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 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 1 1 1 1 1 0 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 1 1 1 1 1 0 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 1 1 1 1 1 0 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 1 1 1 1 1 0 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 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 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 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

Figura A.2: Questão real gerada pelo MCTest para o Simulado 4 — tópico "Operadores Morfológicos".

A.5 Considerações finais

Este apêndice resumiu os passos para instalar o MCTest, criar exames e construir questões parametrizadas com integração ao VPL do Moodle. O uso combinado de variáveis sorteadas na descrição (`[[code:...]]`) e de código Python embutido na própria questão (`[[def: ... ]]`) permite gerar centenas de variações de prova a partir de um único modelo, cada uma corrigida automaticamente e de forma consistente — com a variável `texto`, em particular, garantindo que o mesmo enunciado seja exibido tanto no PDF impresso quanto na avaliação da questão pelo estudante no Moodle. O exemplo real do Simulado 4 (Seção A.4) ilustra como esse mecanismo é usado na prática para gerar questões de programação com cenários visuais distintos por estudante. O Apêndice B detalha como essas variações são publicadas e corrigidas no Moodle, e o Apêndice C descreve como restringir o acesso às provas por meio do SEB.

Apéndice B

Sobre o Moodle (VPL)

Este apêndice descreve a configuração de uma atividade **VPL (*Virtual Programming Lab*)** no Moodle, *plugin* utilizado para a submissão e a correção automática dos Exercícios de Programação (EPs), dos simulados quinzenais e das provas descritas neste livro. A versão do Moodle utilizada foi a **4.5**.

B.1 Configurando uma atividade VPL

A criação de um exame no MCTest com integração ao VPL (ver Apêndice A) gera dois arquivos que devem ser renomeados para `linker.json` e `students_variations.csv`. Esses arquivos, juntamente com o `morph.py`, devem ser adicionados aos **Arquivos de Execução** da atividade VPL, além de todos os arquivos contidos em `VPL_modification/V14-AI-feedback.zip`, disponível no repositório do MCTest em https://github.com/fzampirolli/mctest. Todos eles devem ser marcados com a opção **"*Files to keep when running*"**.

Em síntese, a atividade VPL passa a conhecer, para cada estudante, qual variação da questão foi sorteada (por meio de `students_variations.csv`) e qual é o conjunto de casos de teste correspondente (por meio de `linker.json`). Isso permite que a correção seja individualizada, mesmo quando dezenas de estudantes resolvem a mesma atividade com variações distintas.

B.2 Publicando os EPs como atividades VPL

Os Exercícios de Programação (EPs) apresentados ao final de cada capítulo deste livro também podem ser publicados como atividades VPL, seguindo o mesmo mecanismo. O fluxo típico de uso em sala de aula foi:

1. Abertura dos dois *notebooks* Colab do capítulo (teórico e prático), com destaque para os simuladores interativos;
2. Execução dos blocos de código, alterando parâmetros para reforçar a compreensão dos conceitos;
3. Nas aulas em laboratório, submissão das respostas dos EPs diretamente na atividade VPL correspondente, com retorno imediato sobre a aprovação ou reprovação nos casos de teste.

Caso a atividade VPL não seja gerada pelo MCTest, os arquivos `linker.json` e `students_variations.csv` não serão necessários. Entretanto, para utilizar o *feedback* por IA (ver Apêndice D), devem ser adicionados os arquivos contidos em `VPL_modification/V14-EP0_VPL-TEMPLATE.zip`, disponível no mesmo repositório (https://github.com/fzampirolli/mctest), seguindo o procedimento descrito na Seção B.1.

> **Reaproveitamento dos casos de teste**
>
> Os arquivos `.cases` utilizados na validação local (classe `TestSuite`, descrita no corpo deste livro) são compatíveis com o formato esperado pelo VPL, permitindo reaproveitar o mesmo conjunto de testes tanto no desenvolvimento do EP quanto na correção automatizada no Moodle.

Nos simulados quinzenais — aplicados com o SEB (ver Apêndice C) e com bônus de 5% sobre a nota final —, foram utilizadas atividades VPL com EPs semelhantes aos das listas regulares, porém corrigidas sob restrição de acesso à internet, reforçando o caráter avaliativo da atividade.

B.3 Considerações finais

Este apêndice descreveu a configuração de atividades VPL no Moodle para a correção automática de EPs, simulados e provas parametrizadas. A combinação entre MCTest (geração de variações e casos de teste), VPL (submissão e correção) e SEB (restrição de acesso) forma o tripé sobre o qual se apoia a avaliação automatizada e individualizada utilizada ao longo da disciplina.

Apéndice C

Uso do SEB nos simulados quinzenais e avaliações

Além das duas provas com questões parametrizadas (ver Apêndice A), o ***Safe Exam Browser*** **(SEB)** também foi utilizado para restringir o acesso aos **simulados quinzenais** — atividades VPL (*Virtual Programming Lab*) com Exercícios de Programação (EPs) semelhantes aos das listas regulares, valendo um bônus de até 5% sobre a nota final —, bem como às demais avaliações práticas.

O SEB garante um ambiente seguro de avaliação ao bloquear o acesso a recursos externos não autorizados nas máquinas do laboratório. Para a implementação adotada neste trabalho, utilizou-se o sistema operacional **Windows 10** e a versão **3.7.1.704** do SEB, disponível em https://safeexambrowser.org/, que deve estar previamente instalada em todas as estações do laboratório.

A seguir, descreve-se o procedimento para configurar o arquivo de ambiente do SEB e vinculá-lo à atividade VPL no Moodle.

C.1 Configurando a aplicação no *SEB Tool*

Para gerar o arquivo de configuração `.seb` que será distribuído aos estudantes, abra o utilitário *SEB Tool* e siga os passos descritos a seguir.

1. ***General***
   - No campo ***Start URL***, insira a URL direta da atividade VPL no Moodle.
   - *Nota:* caso seja necessário navegar por mais de uma atividade dentro do mesmo curso, insira a URL principal do curso e, na aba ***Browser***, marque a opção ***Allow navigation back/forward in exam***.
2. ***Network***
   - Inclua as URLs permitidas para navegação. Para restringir o acesso à atividade desejada, utilize uma expressão de filtro no padrão `/mod/vpl/*?id=4624*`, em que o número final corresponde ao identificador da atividade VPL no Moodle. Na Tabla C.1, a última linha contém a URL da página principal do curso (identificador `475`), mas ela também pode ser substituída pela URL de uma seção que contenha várias atividades. Essa configuração é opcional e útil quando há mais de uma atividade VPL. Nesse caso, a primeira linha da tabela deve ser repetida para cada atividade avaliativa que deverá ficar acessível durante o *exame*.
   - **Configurações gerais da aba *Filter***:
     ☒ *Activate URL filtering*
     ☐ *Filter also embedded content*

Tabla C.1: Expressões de filtro de URL utilizadas na configuração do SEB.

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*

Active	*Regex*	*Expression*	*Action*
[x]	[ ]	`https://mctest.ufabc.edu.br/compare`	*Allow*
[x]	[ ]	`https://moodle.ufabc.edu.br/enrol/index.php?id=475`	*Allow*

3. ***Security***
   - Caso o laboratório utilize um projetor ou monitor secundário, ajuste o campo ***Maximum allowed number of connected displays*** para um valor maior que 1, evitando que o SEB bloqueie a exibição.
4. ***Config File***
   - Clique em ***Save Settings As...*** e salve o arquivo com a extensão `.seb`.
5. ***Exam***
   - Marque a opção ***Use Browser Exam Key and Configuration Key***.
   - Copie o código gerado no campo ***Browser Exam Key***. *Importante:* copie essa chave somente **após** salvar o arquivo no passo anterior.

C.2 Vinculando a chave do SEB à atividade VPL no Moodle

Após gerar o arquivo e obter a *Browser Exam Key*, acesse o Moodle para aplicar as restrições.

1. Na página da atividade VPL, clique no ícone de engrenagem (**Configurações**).
2. Acesse ***Configuration*** → ***Submission restrictions*** e clique em ***Show more***.
3. Altere a opção ***SEB browser required*** para ***Yes***.
4. Cole a chave copiada no campo ***SEB exam key(s)***.

C.3 Restrição por endereço IP (opcional)

Para garantir que as submissões ocorram exclusivamente a partir dos computadores do laboratório, é possível combinar o SEB com o filtro de rede do Moodle, preenchendo o campo ***Allowed submission from net***.

- **Faixas consecutivas:** utilize um hífen para definir o intervalo. Exemplo: `111.11.11.1-62` (permite os IPs de `111.11.11.1` a `111.11.11.62`).
- **IPs não consecutivos:** informe os endereços individuais separados por vírgulas.

C.4 Considerações finais

A integração do SEB com o VPL no Moodle cria um fluxo simples para o estudante: basta dar um duplo clique no arquivo `.seb` disponibilizado pelo docente para que o navegador seguro seja aberto diretamente na atividade configurada.

A combinação da *Browser Exam Key* com a restrição por faixa de endereços IP das máquinas do laboratório fornece uma solução robusta para avaliações de programação *online*, dificultando acessos indevidos e contribuindo para a integridade do processo de avaliação.

Apéndice D

Geração de material didático com o *Gemini Notebook*

Este apêndice descreve o uso do ***Gemini Notebook*** (***NotebookLM***) como ferramenta de apoio à preparação do material didático da disciplina, complementando o uso de IA na revisão de textos e códigos, mencionado no prefácio deste livro.

D.1 Vídeos conceituais e de resolução de EPs

Para cada capítulo, foi criado um projeto no *Gemini Notebook* tendo como fontes o PDF completo do livro e o arquivo `morph.py`. A partir dessas fontes, o *Gemini Notebook* gerou automaticamente um vídeo curto (em média, 7 minutos), utilizado na abertura das aulas. Os capítulos alternaram dois tipos de vídeo:

- **Vídeo conceitual**, apresentando os fundamentos teóricos do capítulo, geralmente exibido na primeira aula dedicada ao tema;
- **Vídeo de resolução de EPs**, apresentando uma estratégia de solução para os Exercícios de Programação, exibido na aula seguinte, quando a maior parte dos EPs era resolvida em conjunto com a turma.

> **Papel do vídeo na aula**
>
> O vídeo gerado pelo *Gemini Notebook* não substitui a explicação do professor. Ele funciona como uma breve introdução e motivação ao tema, contextualizando o estudante antes da apresentação dos *slides* e da abertura dos *notebooks Colab*.

D.2 *Slides* de apoio

Complementando os vídeos, o *Gemini Notebook* também gerou, a partir das mesmas fontes (o PDF do livro e o arquivo `morph.py`), um conjunto de aproximadamente 12 *slides* por capítulo, abrangendo tanto os conceitos teóricos quanto a parte prática. A geração utilizou um *prompt* específico para cada capítulo, delimitando o escopo do conteúdo a ser sintetizado e evitando tanto a antecipação de assuntos de capítulos posteriores quanto a reprodução de trechos extensos do livro sem adaptação didática.

Após a apresentação dos *slides*, a aula prosseguia com a abertura dos dois *notebooks Colab* do capítulo (teórico e prático), enfatizando os simuladores interativos e a execução dos blocos de código com alteração de parâmetros.

D.3 Material principal e considerações finais

A geração de vídeos e *slides* com o *Gemini Notebook*, a partir do próprio livro e da biblioteca `morph.py`, permitiu padronizar a abertura das aulas e reduzir o tempo de preparação do material didático, sem prescindir da curadoria do professor na elaboração dos *prompts* e na condução das atividades em sala de aula.

Os vídeos e *slides* produzidos pelo *Gemini Notebook* têm caráter **motivacional** e servem como ponto de partida para a discussão dos conceitos apresentados em cada capítulo. Como qualquer conteúdo gerado por IA, eles podem conter imprecisões ou erros ocasionais. Em alguns casos, essas imprecisões são exploradas intencionalmente durante a aula para verificar se os estudantes estão acompanhando criticamente a apresentação e conseguem identificar inconsistências conceituais.

O conteúdo oficial da disciplina, entretanto, é o material produzido pelo método descrito no prefácio deste livro, disponibilizado em três formatos complementares: **HTML**, para navegação com simuladores interativos; **PDF**, para leitura e impressão; e ***notebooks Jupyter*** (`.ipynb`), para execução e experimentação dos códigos. Todo o material gerado pelo *Gemini Notebook* deve ser entendido como um recurso complementar a esse conteúdo principal.

Assim como nos demais usos de IA generativa descritos neste livro, a responsabilidade pela qualidade técnica, pela correção conceitual e pela adequação pedagógica do material permanece com o professor.

Apéndice E

Uso de IA na Avaliação de Estudantes: o Projeto vpl-ai-feedback

Este apêndice descreve o **vpl-ai-feedback**, sistema de código aberto desenvolvido para apoiar a correção de submissões de código enviadas ao VPL (Moodle) por meio de *feedback* gerado por Inteligência Artificial (IA). Diferentemente de um corretor automático tradicional — que apenas compara saídas com casos de teste —, o **vpl-ai-feedback** envia o código do estudante, junto com a rubrica da questão, a um modelo de linguagem (LLM), que devolve uma análise qualitativa por critério, no estilo de um ***feedback* socrático**: o estudante recebe comentários que apontam o que foi feito corretamente, o que pode ser melhorado e por quê, tanto em atividades **formativas** (exercícios de prática) quanto em atividades **avaliativas** (simulados) do VPL.

Esse mecanismo de *feedback* socrático em atividades VPL foi proposto por Zampirolli et al. (2026). O estudo descreve a integração de IA ao processo de avaliação de exercícios de programação parametrizados no VPL/Moodle, utilizando um conjunto de sete LLMs adotados para analisar o código dos estudantes e gerar *feedback* automatizado a cada solicitação de correção, com resultados preliminares indicando percepção positiva dos estudantes quanto à geração de ideias, clareza das explicações e autonomia de aprendizagem.

O restante deste apêndice detalha a implementação prática dessas ideias no projeto **vpl-ai-feedback** — cujo código está disponível em https://github.com/fzampirolli/vpl-ai-feedback — e descreve especificamente seu uso nas três turmas de PDI-VC entre maio e agosto de 2026.

> ⚠ Atenção: IA não substitui o julgamento docente
>
> O uso de IA para **avaliar** estudantes **não é recomendável como fonte única de nota**, pois modelos de linguagem podem **alucinar** — isto é, produzir avaliações, critérios ou justificativas plausíveis, porém incorretas, mesmo diante de um código correto ou incorreto. Um LLM pode atribuir nota alta a um código com erro sutil, ou nota baixa a um código correto mas escrito de forma incomum, simplesmente porque a explicação gerada "parece" coerente. **A IA deve ser usada apenas como um complemento à avaliação**, nunca como substituta do professor. O Apêndice A e o Apêndice B descrevem os mecanismos de correção objetiva (casos de teste) que continuam sendo a base da nota oficial; este apêndice trata exclusivamente do uso da IA como camada adicional de *feedback* qualitativo.

E.1 Contexto de uso

O **vpl-ai-feedback** foi utilizado em três turmas da disciplina **PDI-VC**, ministradas entre maio e agosto de 2026, totalizando aproximadamente uma centena de estudantes. O *feedback* por IA foi empregado de três formas complementares:

1. ***Feedback* contínuo em atividades formativas** — exercícios de prática enviados ao VPL ao longo do curso, nos quais o estudante podia reenviar o código quantas vezes desejasse. A cada submissão, recebia um *feedback* qualitativo gerado por IA, além da correção objetiva realizada pelo VPL, descrita em Zampirolli et al. (2026).

2. ***Feedback* complementar em simulados (atividades avaliativas bônus)** — quatro simulados realizados ao longo do quadrimestre, previstos no plano de ensino como atividades bônus, cada um

valendo até 5% da nota final. Aplicados aproximadamente 30 minutos antes do término das aulas práticas quinzenais em laboratório, os simulados utilizavam o **vpl-ai-feedback** para gerar uma rubrica individual de avaliação, enviada por e-mail juntamente com um resumo comparativo entre a nota atribuída pelo Moodle/VPL e a avaliação produzida pela IA.

3. ***Feedback* complementar nas Provas 1 e 2 (avaliações oficiais)** — as duas provas formais do quadrimestre, cada uma valendo uma parcela maior da nota final da disciplina, também passaram a utilizar o **vpl-ai-feedback** após sua aplicação. Diferentemente dos simulados (bônus), aqui a nota oficial já é definida pela correção objetiva do VPL e/ou avaliação manual do professor antes da divulgação; a rubrica gerada pela IA é enviada ao estudante apenas posteriormente, como material de apoio para revisão do próprio desempenho — reforçando, também nas avaliações de maior peso, a separação entre *nota oficial* e *feedback* complementar detalhada na Seção D.7.

Uma pesquisa de opinião, respondida por 102 estudantes antes da adoção do sistema, indicou elevada aceitação da proposta. Apenas 2 estudantes atribuíram nota 1 (rejeição) e 7 atribuíram nota 2 ao interesse em receber *feedback* automático de código por IA. Outros 19 declararam-se indiferentes (nota 3), enquanto a maioria — 38 e 37 estudantes — atribuiu notas 4 e 5 (alta aprovação), respectivamente, totalizando os 102 respondentes. Esse resultado motivou a adoção do sistema como complemento ao processo formativo, e não como substituto da correção humana.

! A nota oficial nunca foi a nota da IA

É fundamental destacar que, conforme definido no plano de ensino, **a nota utilizada tanto para o cálculo do bônus de cada simulado quanto para a nota oficial das Provas 1 e 2 foi sempre a nota atribuída pelo VPL** (baseada nos casos de teste) e/ou pela avaliação manual do professor — e **não** a nota sugerida pela IA. A rubrica gerada pelo **vpl-ai-feedback** foi enviada ao estudante apenas como material de apoio ao aprendizado — nunca como critério de atribuição de nota, seja em atividades bônus ou em avaliações oficiais. Essa separação entre *nota oficial* (VPL/professor) e *feedback complementar* (IA) é o princípio central deste apêndice e foi retomada na Seção D.8.

E.2 Arquitetura do projeto

O **vpl-ai-feedback** é um *script* Python assíncrono organizado da seguinte forma:

```
.
  config.yaml              ← configuração (credenciais, provedor, pesos) - NUNCA versionado
  config.yaml.example      ← template público de configuração
  main.py                  ← script principal (async), orquestra todo o processo
  run.sh                   ← wrapper de execução (bash run.sh config.yaml)
  gerar_relatorio.py       ← converte o consolidado *_ALL.txt em CSV
  enviar_email.py          ← envia as rubricas por e-mail a cada estudante
  providers/                ← clientes das APIs de LLM
      __init__.py          ← fábrica de clientes (Factory)
      base.py               ← lógica de retry, backoff e fallback entre modelos
      groq.py
      deepseek.py
      gemini.py
  core/                      ← lógica de negócio
      grader.py              ← identifica arquivos por questão, monta o prompt, chama a LLM
      utils.py
  <pasta_da_turma>/          ← submissões baixadas do Moodle VPL
      Nome estudante - login/
          <timestamp>/                 ← última submissão do estudante
              Q1.py                    ← padrão para provas com múltiplas questões
              Q2.py
              rubrica.txt              ← gerado pela LLM (somente se avaliação válida)
```

```
        <timestamp>.ceg/
            execution.txt            ← nota/objetiva atribuída pelo VPL
    ...
```

O sistema aceita três provedores de LLM — **Groq**, **DeepSeek** e **Gemini** — configuráveis em `config.yaml` por meio da chave `llm.provider`. Cada provedor pode ter **múltiplos modelos** cadastrados; a cada chamada, a lista é **embaralhada**, distribuindo a carga entre os estudantes processados em paralelo e reduzindo a chance de erros de limite de requisições (HTTP 429). Quando um modelo falha, o sistema tenta até três vezes antes de passar ao próximo da lista, respeitando o tempo de espera sugerido pelo provedor. Erros de autenticação (401) ou de saldo insuficiente (402) interrompem imediatamente a tentativa naquele provedor.

Um aspecto importante do projeto diz respeito à privacidade dos dados. **Nome, e-mail, login, RA e CPF do estudante nunca são enviados à API do LLM**. São transmitidos para processamento apenas: o nome do arquivo contendo o código-fonte; o próprio código (com os comentários removidos); o *prompt* da rubrica — que não contém dados pessoais —; e, quando disponíveis no `execution.txt` gerado pelo VPL, o enunciado oficial da questão sorteada para aquele estudante e o resultado bruto dos casos de teste executados (entrada, saída produzida pelo código e saída esperada). Esses dois últimos itens, embora extraídos de um arquivo específico de cada estudante, também não contêm identificação pessoal — apenas o texto da questão e os valores de entrada/saída dos testes automáticos —, e são usados como evidência objetiva para reduzir o risco de a IA identificar incorretamente o tipo de questão ou avaliar a lógica do código apenas por leitura estática (Seção D.3).

Ainda assim, é importante reconhecer que os três provedores atualmente suportados — Groq, DeepSeek e Gemini — são serviços comerciais operados por empresas estrangeiras, cujos modelos são executados em infraestrutura fora do país e sujeitos a políticas de uso, disponibilidade e custo que fogem ao controle da instituição de ensino. Essa dependência de provedores externos traz riscos que vão além da correção de código pontual: mudanças de preço, descontinuação de modelos, indisponibilidade temporária do serviço, alterações unilaterais nos termos de uso e, em casos mais sensíveis, restrições geopolíticas de acesso podem comprometer a continuidade de um sistema de avaliação que passe a depender estruturalmente dessas APIs. Por isso, é estratégico que Instituições de Ensino Superior (IES) como a UFABC invistam no desenvolvimento e na hospedagem de **provedores próprios de IA** — por exemplo, modelos abertos (*open-weight*) executados em infraestrutura local ou em nuvem soberana —, reduzindo a dependência de serviços externos, especialmente de outros países, e ampliando o controle institucional sobre custos, disponibilidade, privacidade dos dados de estudantes e continuidade pedagógica de longo prazo. O desenho do **vpl-ai-feedback** já favorece esse caminho: a arquitetura de `providers/` foi construída como uma fábrica (*factory*) extensível, na qual um novo provedor — inclusive um modelo hospedado localmente pela própria instituição, com interface compatível — pode ser adicionado com baixo esforço de integração.

E.3 O *prompt* universal e a rubrica em duas etapas

Cada tipo de prova é descrito em um arquivo de *prompt* (por exemplo, `Simulado4.txt`), referenciado em `grading.prompt_file` no `config.yaml`. Esse arquivo instrui a LLM a realizar a avaliação em **duas etapas**:

1. **Identificação da questão** — a LLM determina o tipo específico de operação sorteado para aquele estudante (útil quando há questões paramétricas sorteadas e embaralhadas por estudante, como descrito no Apêndice A);
2. **Aplicação da rubrica correspondente** — a LLM avalia o código segundo critérios pontuados, cada um com uma faixa máxima de pontos, retornando ao final uma nota no formato `NOTA FINAL: X/PESO`.

Em provas parametrizadas, o arquivo de *prompt* costuma conter **várias rubricas paralelas**, uma para cada tipo de questão possível, delimitadas por marcadores como `[START_RUBRICA_TIPO_A]` / `[END_RUBRICA_TIPO_A]`. A LLM identifica primeiro qual tipo foi sorteado para aquele estudante e aplica **apenas** a rubrica correspondente, ignorando as demais — o que evita que critérios de um tipo diferente contaminem a nota.

Quando a prova possui mais de uma questão por atividade VPL, os arquivos de código devem seguir o

padrão `Q1.*`, `Q2.*` etc. — um arquivo por questão, com extensão livre conforme a linguagem escolhida pelo estudante —, e cada questão tem peso configurável individualmente em `grading.weights` no `config.yaml`. Quando a prova tem uma única questão e não foi gerada pelo MCTest, o sistema aceita qualquer nome de arquivo com extensão suportada, desde que seja o único arquivo de código presente na pasta de submissão.

No caso do Simulado 4 (tópico *Operadores Morfológicos*), por exemplo, um dos tipos possíveis definia três critérios, com peso total de 100 pontos para aquela questão:

Critério	Descrição	Pontuação máxima
1 — Entrada de dados	Leitura de `H`, `W` e da matriz binária (`mm.readImg`)	25 pts
2 — Processamento morfológico	Abertura + Fechamento (`mm.sebox()`), `mm.measure`, ordenação por `bbox[0]`/`bbox[1]` e reatribuição de IDs	50 pts
3 — Saída e formatação	Exibição da imagem limpa (`mm.drawImg`) e tabela de medidas formatada	25 pts

O *prompt* exige ainda um **formato de resposta obrigatório** (largura de linha, ordem das seções, marcação `NOTA FINAL:`), o que torna a extração automática da nota e a montagem do relatório consolidado mais confiáveis — embora, como discutido na Seção D.8, isso não elimine o risco de erro semântico na avaliação do conteúdo.

E.4 Duas camadas adicionais de evidência

Para reduzir o risco de a LLM alucinar o tipo de questão ou a corretude do código — o risco central discutido na Seção D.7 —, o **vpl-ai-feedback** passou a fornecer à IA duas fontes de evidência objetiva, extraídas automaticamente do próprio `execution.txt` gerado pelo VPL, além do código-fonte:

- **Enunciado oficial da questão.** Como as questões costumam ser sorteadas e embaralhadas por estudante no MCTest (Apêndice A), o `execution.txt` de cada estudante já traz, no campo "Descrição resumida da questão", o texto exato da questão que caiu para ele. O sistema extrai esse trecho automaticamente e o envia à LLM como evidência **primária** para identificar o tipo/variação da questão — mais confiável do que inferir apenas pela leitura estática do código, especialmente em submissões incompletas ou ambíguas entre dois tipos próximos (por exemplo, erosão "plana" *versus* "com pesos"). Quando o código entregue diverge do que o enunciado pede, o sistema não ignora o enunciado nem "corrige" a leitura silenciosamente: mantém a identificação pelo enunciado, reduz a confiança da avaliação para "Baixa" e explicita a divergência no *feedback* enviado ao estudante.
- **Resultado real da execução no VPL.** Quando disponível, o sistema também envia à LLM os casos de teste efetivamente executados sobre aquele código — entrada, saída produzida e saída esperada —, como evidência objetiva de corretude, a ser usada para confirmar ou refutar a leitura da lógica antes de pontuar os critérios de processamento e de saída, em vez de a IA depender apenas de simular mentalmente o código (o que é especialmente falho em laços com `min`/`max`, deslocamento de índice ou cálculo de limiar de Otsu, onde erros sutis passam despercebidos numa leitura estática).

Além disso, a LLM é instruída a declarar explicitamente seu **nível de confiança** (Alta, Média ou Baixa) na identificação de cada tipo de questão, com justificativa quando a confiança não for Alta. Esse valor é extraído automaticamente e alimenta uma coluna `Revisar_Manualmente` no relatório CSV consolidado (Seção D.4), permitindo ao professor priorizar exatamente os casos em que a própria IA reconhece maior incerteza — em vez de tratar todas as avaliações de uma turma como igualmente confiáveis.

E.5 Fluxo de execução

A execução típica, feita por `bash run.sh config.yaml`, segue os passos:

1. Carrega `config.yaml` e localiza a pasta de submissões (`paths.student_base_dir`);
2. Para cada estudante, identifica a **última submissão** (pasta com o *timestamp* mais recente);
3. Extrai a nota objetiva do Moodle a partir de `*.ceg/execution.txt`;
4. Para cada questão da prova, localiza o arquivo de código (`Q1.*`, `Q2.*`, ou o único arquivo, no caso de prova que não foi gerada pelo MCTest), monta o *prompt* (código + rubrica) e envia a um modelo sorteado da lista configurada;
5. Processa todos os estudantes **em paralelo**, com controle de concorrência;
6. Gera, por estudante, o arquivo `rubrica.txt` — **somente se a IA retornar ao menos uma avaliação válida** para as questões que possuem código; caso contrário, o arquivo não é salvo, forçando nova tentativa na próxima execução;
7. Consolida todos os `rubrica.txt` em um único `*_ALL.txt` e gera um relatório `*_relatorio.csv`, comparando nota do Moodle, nota da IA e a diferença entre ambas, por questão e no total.

Situação	`rubrica.txt` é salvo?
Todas as questões com código avaliadas com sucesso	✅ Sim
IA falhou em ao menos uma questão com código	❌ Não — reprocessa na próxima execução
estudante sem nenhum arquivo de código enviado	❌ Não
Tipo de questão identificado como DESCONHECIDO	❌ Não

E.6 Exemplo ilustrativo: rubrica de uma prova com três questões

ℹ Sobre a origem deste exemplo

O trecho a seguir **não reproduz o código de nenhum estudante específico**. Trata-se de um exemplo reconstruído pelo autor, que reproduz um *padrão* de erro observado repetidamente ao longo das turmas de PDI-VC — a confusão entre uma função morfológica 2D (`mm.ero`/`mm.ero0`) e uma operação 1D sobre sinais — sem citar ou identificar qualquer submissão real. Essa opção evita qualquer risco de violação do direito autoral do estudante sobre seu próprio código-fonte, ainda que anonimizado, já que a titularidade da obra permanece do autor original mesmo quando nome e *login* são removidos.

O trecho a seguir ilustra a saída gerada pelo **vpl-ai-feedback** para um estudante de uma prova com **três questões** (Q1, Q2, Q3), usando o modelo `deepseek-chat`. O resumo comparativo aparece no topo do arquivo, seguido do enunciado oficial de cada questão, do código submetido e da avaliação por critério.

```
RESUMO - IA (deepseek-chat)  x  MOODLE

Peso total : 100 pts

Q1 (IA) :  3.3 / 33 pts
Q2 (IA) :  6.7 / 33 pts
Q3 (IA) : 33.3 / 34 pts

Moodle : (Q1=33 + Q2=0 + Q3=34) = 67 pts
IA     : 43.3 / 100 pts

Diferença (IA - Moodle): -23.7 pts

Q1: tipo identificado = B  | confiança = Baixa (código não
```

```
    corresponde ao enunciado da questão)  REVISAR
  Q2: tipo identificado = A  | confiança = Alta (enunciado oficial
    confirma o tipo e a variação)
  Q3: tipo identificado = C1 | confiança = Alta (enunciado oficial
    confirma erosão 2D com OpenCV integrada)
```

Q1 — código não corresponde ao enunciado (confiança baixa): o enunciado oficial da questão, extraído do `execution.txt`, pedia uma erosão morfológica **1D** sobre um sinal (leitura de N, M, o sinal e o elemento estruturante B, seguida do mínimo dos vizinhos ativos). O código submetido, no entanto, lê apenas dois inteiros e invoca `mm.ero0` — uma função de erosão **2D** para imagens, não relacionada à operação pedida.

```
Tipo identificado: TIPO B (Erosão 1D - plana)
Confiança: Baixa - código não corresponde ao enunciado da questão (usa
mm.ero0, função de erosão 2D, em vez de implementar erosão 1D plana
sobre um sinal)

Critério 1 - [3.33/10.00 pts]:
- O código lê dois inteiros em uma única linha (que corresponderiam a
  N e M), mas não lê o sinal nem o elemento estruturante B como vetores
  1D. Em vez disso, trata ambos como imagens via mm.readImg, o que não
  corresponde à leitura esperada de um sinal 1D e de B em linhas
  separadas.

Critério 2 - [0.00/16.67 pts]:
- A operação aplicada é mm.ero0(img, element), erosão 2D para imagens,
  não a erosão 1D de sinais pedida no enunciado. A lógica de mínimo dos
  vizinhos ativos com bordas por recorte não está implementada.

Critério 3 - [0.00/6.67 pts]:
- A saída é uma imagem 2D via mm.drawImg, não o sinal 1D resultante em
  uma única linha com valores inteiros separados por espaço, como
  exigido.

NOTA FINAL: 3.33/33.33

Comentários para o estudante: O código apresentado não corresponde ao que o
enunciado pede. A questão sorteada solicita uma erosão morfológica 1D
sobre um sinal unidimensional; o código usa uma função de erosão 2D
para imagens. Releia atentamente o enunciado sorteado antes de escolher
a implementação.
```

Q2 — erro de sintaxe impede a execução (confiança alta, nota baixa por falha real): aqui a IA identificou corretamente o tipo (limiarização com Otsu), com apoio do enunciado oficial, mas o código continha um erro de indentação que impedia a execução — evidenciado tanto pela leitura estática quanto pela ausência de saída válida nos testes do VPL.

```
Tipo identificado: TIPO A (Limiarização - T manual + Otsu)
Confiança: Alta - enunciado oficial confirma o tipo e a variação

Critério 1 - [6.67/10.00 pts]:
- Lê corretamente os três parâmetros L, C e T em linhas separadas e a
  imagem de entrada. Porém, a definição de uma função auxiliar de
  limiarização está indentada incorretamente dentro do escopo global,
  causando um erro de indentação que impede a execução completa do
  programa.

Critério 2 - [0.00/16.67 pts]:
- A binarização com limiar fixo T está corretamente estruturada, mas o
  cálculo do limiar de Otsu depende da função auxiliar malformada, que
```

```
  referencia uma variável fora de escopo. O erro de indentação impede
  qualquer processamento.

Critério 3 - [0.00/6.67 pts]:
- Devido ao erro de indentação, nenhuma saída é produzida.

NOTA FINAL: 6.67/33.33

Comentários para o estudante: A estrutura geral está correta (leitura de L,
C, T e da imagem), mas há um erro de indentação que impede a execução.
Revise a sintaxe Python antes de submeter e teste localmente o código
antes do envio ao VPL.
```

Q3 — divergência de função, mas equivalência funcional confirmada por evidência real (confiança alta, nota justa): o enunciado pedia explicitamente o uso de `mm.ero` (versão integrada ao OpenCV), mas o código usa `mm.ero0` (versão plana implementada manualmente). Como o `execution.txt` mostrava os 10 casos de teste do VPL passando integralmente, a IA não penalizou a divergência de função — reconheceu a equivalência funcional demonstrada pelos testes reais, mas ainda assim registrou a observação no *feedback*.

```
Tipo identificado: TIPO C1 (Erosão 2D - plana sem pesos)
Confiança: Alta - enunciado oficial confirma erosão 2D com OpenCV
integrada (mm.ero), e o código implementa erosão plana via mm.ero0

Critério 1 - [10.00/10.00 pts]:
- Lê corretamente altura, largura, a matriz da imagem e as dimensões e
  valores do elemento estruturante B.

Critério 2 - [16.67/16.67 pts]:
- O enunciado pedia o uso de mm.ero (OpenCV integrada), mas o código
  usa mm.ero0 (plana sem pesos). Apesar da divergência de função, a
  operação foi implementada corretamente com o elemento estruturante
  lido: os 10 testes executados no VPL passaram integralmente (10/10),
  confirmando equivalência funcional para os casos testados.

Critério 3 - [6.67/6.67 pts]:
- Imprime a imagem resultante no formato correto; os testes confirmaram
  a saída correta.

NOTA FINAL: 33.34/33.33

Comentários para o estudante: O código está correto e funcional, com todos
os 10 testes passando. Observação: o enunciado pedia explicitamente
mm.ero (versão OpenCV), mas você usou mm.ero0 (versão plana). Embora os
resultados tenham sido equivalentes nos testes, siga exatamente a
função solicitada no enunciado em provas futuras.
```

Note que, nesse caso composto, a IA atribuiu **23,7 pontos a menos** do que a nota objetiva do VPL, puxada principalmente pela Q1. A diferença agregada por si só já seria um sinal para investigação (Seção D.7), mas o próprio sistema entrega ao professor a causa provável de cada questão: a coluna `Revisar_Manualmente` do CSV é marcada "SIM" sempre que ao menos uma questão do estudante recebe confiança baixa, com o motivo específico registrado na coluna de confiança correspondente — reduzindo o tempo que o professor precisa gastar para localizar, entre uma centena de estudantes, os poucos casos que realmente merecem atenção manual antes da atribuição da nota oficial. O caso da Q3, por sua vez, ilustra o cuidado inverso: a divergência entre enunciado e código **não** resultou em penalização indevida, porque a evidência real de execução confirmou a equivalência funcional — exatamente o comportamento que a Seção D.3 descreve para essas duas camadas de evidência.

E.7 Envio do *feedback* por e-mail

Após a geração das rubricas, o *script* `enviar_email.py` envia a cada estudante um e-mail com o `rubrica.txt` em anexo. O corpo do e-mail é definido em `config.yaml`, em `templates.corpo`, e é interpolado com `{nome_pasta}` e `{login}`:

```yaml
email:
  smtp_server: smtp.ufabc.edu.br
  smtp_port: 587
  from_address: professor@ufabc.edu.br
  password: "SUA_SENHA_AQUI"     # nunca versionar credenciais reais
  use_tls: true

templates:
  assunto: "Rubricas e Correções geradas por LLM - Simulado - {login}@aluno.ufabc.edu.br"
  corpo: |
    Prezado(a) {nome_pasta},
    ...
```

! Segurança de credenciais

O `config.yaml` contém segredos reais (senha de SMTP, chaves de API). Ele deve constar do `.gitignore` do projeto e **jamais** ser publicado, compartilhado ou versionado — inclusive em anexos de e-mail, capturas de tela ou repositórios públicos.

Um exemplo real de e-mail enviado a um estudante (dados **anonimizados**, com nome e login substituídos por um caso fictício) é reproduzido a seguir, para ilustrar o tom didático e as ressalvas explicitadas ao estudante:

! Exemplo de mensagem individual (anonimizada)

Prezado(a) [Nome do Estudante] - [login],
A sua nota do Simulado 4 está disponível no Moodle.
Envio abaixo as competências avaliadas e uma correção detalhada do seu código, gerada automaticamente por Inteligência Artificial (IA).
No anexo "rubrica.txt" está o retorno completo da IA (recomendo baixar e abrir com um editor de texto ou bloco de notas).
Ressalto que essa correção gerada por IA PODE conter imprecisões ou erros, mas serve como um excelente apoio ao seu processo de aprendizagem na disciplina.
Uma sugestão pedagógica é submeter os seus códigos (contidos neste arquivo), juntamente com a RUBRICA abaixo, a outros modelos de LLM para comparação. Isso pode ajudar a identificar possíveis divergências, além de oferecer diferentes perspectivas sobre o seu código e sobre os critérios de avaliação.
Caso tenha dúvidas ou note alguma inconsistência gritante na correção apresentada no Moodle, estou à disposição para esclarecimentos.
Atenciosamente,
Prof. Francisco Zampirolli
PS.: Explicando o processo: a última versão do seu código salva no Moodle foi anexada ao prompt e enviada para um dos LLMs escolhidos de forma integrada ao nosso sistema de avaliação (modelos = ("deepseek-chat")). O processo é repetido até que seja obtida uma resposta com nota válida (entre 0 e 100).

Três elementos didáticos merecem destaque nesse texto: (i) o alerta explícito de que a correção **pode conter erros**; (ii) o convite para que o próprio estudante **contraste a avaliação com outros LLMs**, transformando a IA em ferramenta de estudo ativo em vez de veredito passivo; e (iii) a explicação transparente do processo automatizado, incluindo o(s) modelo(s) usado(s) e a política de reprocessamento até obter uma nota válida.

E.8 Riscos, limites éticos e responsabilidade docente

! O professor não pode simplesmente adotar a nota da IA

Este é o ponto central deste apêndice: **em nenhuma hipótese a nota sugerida pela IA deve ser atribuída automaticamente ao estudante como nota oficial**. Modelos de linguagem podem alucinar — atribuir pontos a critérios não atendidos, "inventar" que uma função foi chamada corretamente quando não foi, ou penalizar um código correto por má interpretação da rubrica. A nota final de qualquer atividade avaliativa deve **sempre** resultar de avaliação manual do professor (ou da correção objetiva por casos de teste, como no VPL/MCTest), com a IA cumprindo, no máximo, o papel de **primeira leitura** ou de **material de apoio ao estudante**, nunca de instância decisória.

Alguns cuidados práticos adotados no uso do **vpl-ai-feedback**, e recomendados a qualquer professor que deseje adaptar o sistema:

- **Separação clara entre nota oficial e *feedback* de IA.** No caso relatado, a nota do bônus dos simulados sempre veio do VPL, nunca da IA (Seção D.1). A rubrica de IA foi comunicada ao estudante como material de apoio, com aviso explícito de que pode conter erros.
- **Transparência com o estudante.** O e-mail enviado (Seção D.6) explicita que a avaliação foi gerada por IA, informa o(s) modelo(s) utilizados e convida o estudante a comparar com outros LLMs — reduzindo a assimetria de informação e incentivando o pensamento crítico sobre a própria correção recebida.
- **Reprocessamento em vez de *cache* inválido.** O sistema só grava `rubrica.txt` quando obtém avaliação válida (Seção D.4); isso evita que uma resposta malformada, incompleta ou visivelmente inconsistente da IA seja apresentada ao estudante como se fosse definitiva.
- **Monitoramento das divergências.** O relatório CSV consolidado (coluna `Diferenca = Total_IA - Total_Moodle`) permite ao professor identificar rapidamente os casos em que IA e correção objetiva mais discordam — como no exemplo da Seção D.5 (+20 pontos) —, priorizando esses casos para revisão manual.
- **Canal aberto para contestação.** O e-mail final oferece explicitamente ao estudante a possibilidade de reportar inconsistências, mantendo o professor como autoridade final sobre a nota.
- **Dados pessoais fora do *prompt*.** Como descrito na Seção D.2, nome, e-mail, login, RA e CPF nunca são enviados à API do LLM, reduzindo riscos de privacidade no uso de provedores externos de IA.

E.9 Considerações finais

O **vpl-ai-feedback** mostra como a IA pode enriquecer o ciclo de *feedback* em disciplinas de programação com alto volume de submissões, oferecendo a cada estudante uma leitura qualitativa e individualizada do próprio código — algo inviável de se fazer manualmente para uma centena de estudantes, em múltiplas questões e provas ao longo do quadrimestre. A pesquisa de opinião citada na Seção D.1 sugere que esse tipo de retorno é bem recebido pelos estudantes. Ainda assim, o uso responsável do sistema depende de um princípio inegociável, reiterado ao longo deste apêndice: **a IA complementa, mas não substitui, a avaliação do professor**, e a nota oficial de qualquer atividade avaliativa deve continuar sendo definida por correção objetiva (VPL/MCTest, Apêndices A e B) e/ou julgamento humano — nunca pela nota bruta devolvida por um modelo de linguagem.

EL VIAJE DE PDI & VC

SISTEMATIZANDO EL CONOCIMIENTO, DESDE EL PÍXEL HASTA LA INTELIGENCIA

Parte I — Fundamentos de PDI

- 1. Fundamentos y primeros pasos
- 2. Muestreo, cuantización y conectividad
- 3. Operaciones espaciales y filtrado
- 4. Morfología matemática y segmentación
- 5. Transformadas y compresión

Parte II — Visión por Computadora

- 6. Análisis e inspección de documentos
- 7. Clasificación y reconocimiento de patrones
- 8. Comprensión de escenas y segmentación
- 9. Aprendizaje profundo

Apéndices esenciales

- **A:** MCTest
- **B:** Moodle/VPL
- **C:** SEB/Simuladores
- **D:** Gemini/Material didáctico
- **E:** IA en el vpl-ai-feedback
- **F:** Percepción y evaluación

IMAGEN DE ENTRADA (PÍXEL) → **PDI** (PROCESAMIENTO) → **VC** (VISIÓN) → SALIDA INTELIGENTE (SEMÁNTICA)

DOMINA LOS PÍXELES, CONSTRUYE EL FUTURO.

Desde histogramas simples hasta la comprensión compleja de escenas con aprendizaje profundo, este trabajo ofrece una hoja de ruta completa. Transforma imágenes en datos y datos en decisiones inteligentes.

Tu capacidad para dar forma a lıa percepción computacional comienza aquí. ¡Mantente a la vanguardia, explora e innova!

Disponible en formato digital (HTML, PDF y Jupyter Notebook), con simuladores y ejercicios, incluidos casos de prueba con VPL y Moodle.

ESCANEA Y ACCEDE A CONTENIDO EXTRA

AUTOR: FRANCISCO DE ASSIS ZAMPIROLLI

https://fzampirolli.github.io/dip-cv/