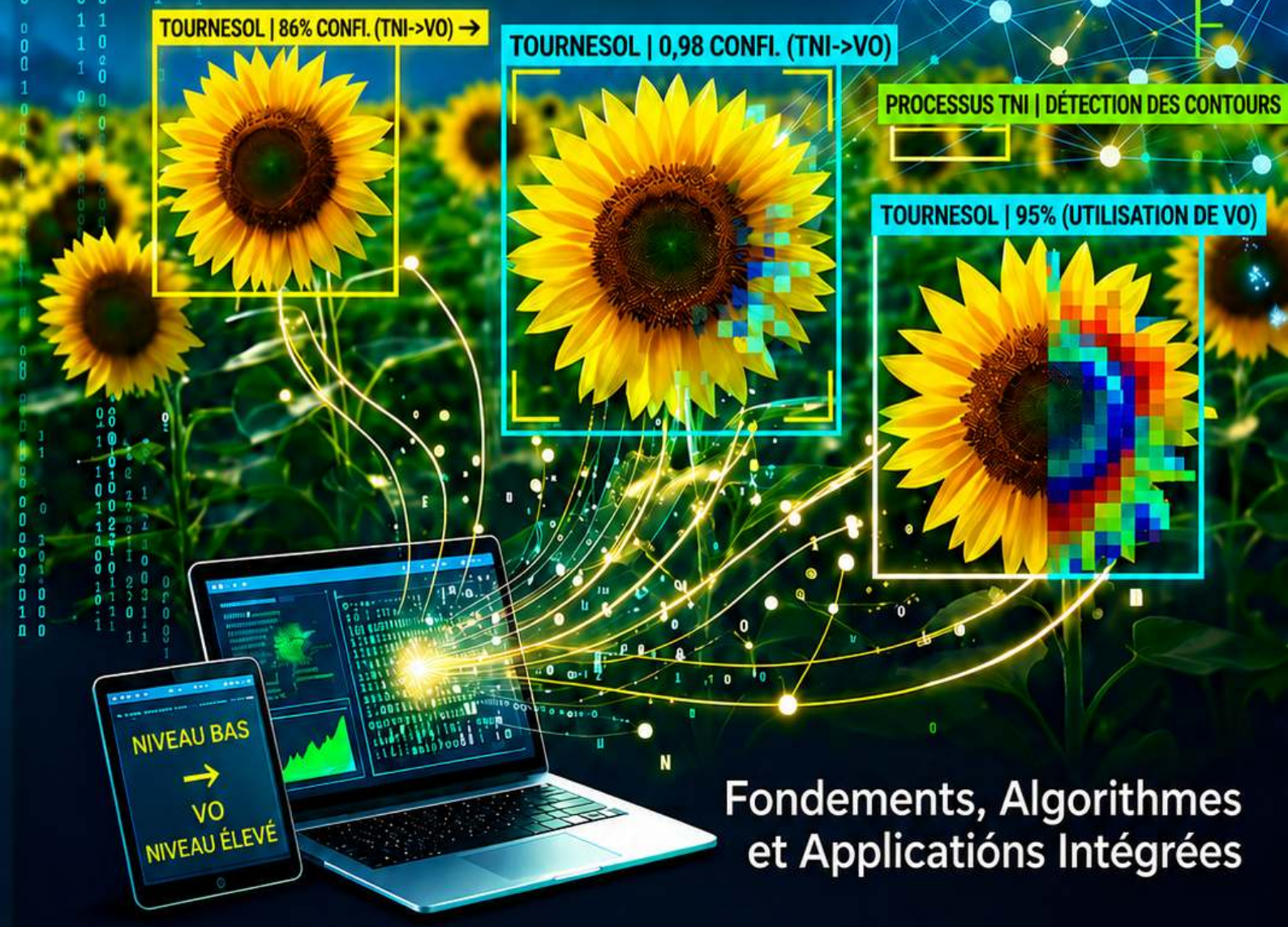
TNI & VO

Traitement Numérique des Images et Vision par Ordinateur



Fondements, Algorithmes et Applicatións Intégrées

AUTEUR: FRANCISCO DE ASSIS ZAMPIROLLI

Traitement Numérique d'Images & Vision par Ordinateur

Livre interactif avec Python

Francisco de Assis Zampirolli

Universidade Federal do ABC

20 septembre 2026

© 2026 Francisco de Assis Zampirolli de l'Université Fédérale de l'ABC (UFABC).
Tous droits réservés.

Ce livre est sous la licence :
Creative Commons Attribution-ShareAlike 4.0 International License
Détails à l'adresse : creativecommons.org/licenses/by-sa/4.0

Conception graphique : Francisco de Assis Zampirolli
Couverture : Francisco de Assis Zampirolli, avec le soutien de l'IA (Gemini et ChatGPT)

CATALOGAGE À LA SOURCE
SYSTÈME DES BIBLIOTHÈQUES DE L'UNIVERSITÉ FÉDÉRALE DE L'ABC

[CODE CUTTER] Zampirolli, Francisco de Assis
Traitement Numérique d'Images et Vision par Ordinateur / Francisco de Assis Zampirolli – Santo André, SP : Édition de l'Auteur, 2026.

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

ISBN : [ISBN À DÉFINIR] (1ère Édition)

1. Traitement Numérique d'Images. 2. Vision par Ordinateur. 3. Morphologie Mathématique. 4. Python. 5. Correction Automatique. 6. Moodle. I. Titre.

CDD [XX] éd. – [CODE CDD]

Table des matières

TNI & VO — Python 1
Organisation . . . 1

Préface 3
Contexte de la première édition . . . 3
Comment ce livre est produit . . . 4
Utilisation d'outils d'Intelligence Artificielle . . . 5
La bibliothèque `morph.py` . . . 6
Exercices de Programmation (EPs) et Validation Automatique . . . 7
Code ouvert . . . 7
Avant de commencer : *Notebooks* en Python . . . 8
Guide de l'Enseignant : Publication des EPs sur Moodle . . . 8
Intégration avec Moodle (VPL) . . . 8
Comment publier sur Moodle . . . 9

I Partie I — Fondements du TNI 1

1 Fondements et Premiers Pas 3
1.1 Objectifs . . . 3
1.2 Avant de commencer : *Notebooks* interactifs . . . 3
1.3 Fondements . . . 3
1.3.1 👁️ Vision par ordinateur . . . 4
1.3.2 🖼️ Traitement Numérique des Images (TNI) . . . 4
1.4 Étapes du PDI . . . 5
1.5 Formation de l'image et le spectre . . . 5
1.6 Qu'est-ce qu'une image numérique ? . . . 7
Représentation mathématique . . . 7
1.7 Configuration de l'environnement . . . 9
1.8 Versions de `morph.py` . . . 9
1.9 Fondements des matrices — attention à la copie de références . . . 10
1.10 Lecture et Affichage d'Images . . . 11
Alternative : *Téléchargement* de l'image vers un stockage local . . . 12
1.11 Conversion de Types et Seuillage . . . 13
Conversion en Niveaux de Gris . . . 13
Seuillage (*Thresholding*) . . . 13
Exemple pratique de conversion et de seuillage . . . 14
1.12 Limiarisation par la méthode d'Otsu . . . 14
1.13 Accès aux pixels . . . 16
1.14 Résumé . . . 17
1.15 🤖 Utilisation du Notebook Gemini comme tuteur complémentaire . . . 18
Fonctionnalités Disponibles sur la Plateforme . . . 18
1.16 Liste d'exercices . . . 19
Références du chapitre . . . 19
1.17 **Calculs de base en Python** . . . 19
1.18 **Fonction print()** . . . 20

1.19 💻 **Partie Pratique avec Exercices de Programmation** . . . 20
🎯 Objectif de ce Cahier . . . 20
§ Instructions Étape par Étape . . . 21
⚠️ Important : Règles et Bonnes Pratiques . . . 22
1.19.1 EP01_01 📏 Trois métriques de distance en TNI . . . 22
1.19.2 EP01_02 📊 Performance Prédictive — Métriques de ML en VC . . . 29
1.19.3 EP01_03 ↗ *Mean Average Precision* (mAP) — Courbe Précision-Sensibilité . . . 31
1.19.4 EP01_04 🖼️ Lecture et Informations d'une Image sous forme de Matrice . . . 34
1.19.5 EP01_05 🔄 Négatif d'une Image en Niveaux de Gris . . . 37
1.19.6 EP01_06 🎨 Conversion RGB → Niveaux de gris (ITU-R BT.601) . . . 38
1.19.7 EP01_07 ⚫ Seuillage Manuel : Image Binaire . . . 40
1.19.8 EP01_08 🎨 Remappage par Plage d'Intensité . . . 41
1.19.9 EP01_09 🏁 Modèle de Damier : Matrice d'Échecs . . . 43
1.19.10 EP01_10 📄 Métadonnées : Lecture de fichier PGM . . . 45
1.19.11 EP01_11 ↗ Analyse de voisinage : Filtre de maximum 1D . . . 46

2 De la Capture au Pixel - Échantillonnage, Quantification et Connectivité **49**
2.1 Objectifs . . . 49
2.2 L'Œil et la Caméra - Éléments de la Perception Visuelle . . . 49
2.2.1 Vision humaine . . . 49
2.2.2 Capteurs numériques . . . 49
2.3 Illusions d'optique : Les défis de la perception visuelle . . . 50
2.3.1 Ambiguïté et contexte . . . 50
2.3.2 Luminosité et contraste local . . . 51
2.3.3 Géométrie et perspective . . . 51
2.4 Le Modèle Mathématique de la Formation de l'Image . . . 51
2.4.1 Représentation numérique et canaux spectraux . . . 52
2.4.2 Préparation de l'environnement pratique . . . 53
2.4.3 Implémentation de la Lecture d'Image dans `morph.py` . . . 53
2.4.4 RGB et RGBA : Exemple pratique avec l'image Mandrill . . . 54
2.5 Numérisation : Échantillonnage et Quantification . . . 55
2.5.1 Échantillonnage — Discrétisation de l'espace . . . 55
2.5.2 Quantification — Discrétisation de l'intensité . . . 55
2.5.3 Effets de l'échantillonnage et de la quantification . . . 56
2.5.4 Analyse technique . . . 58
2.6 Relations entre Pixels - Topologie de l'Image . . . 59
2.6.1 Voisinage . . . 59
2.6.2 Adjacence, connectivité et chemins . . . 60
2.6.3 Distances entre pixels . . . 60
2.7 Stockage d'images . . . 60
2.7.1 Exemple : Extraction de métadonnées et localisation GPS . . . 61
2.7.2 Pourquoi cette séparation est-elle importante ? . . . 63
2.8 Transformations géométriques de base . . . 64
2.8.1 Translation . . . 64
2.8.2 Rotation . . . 65
2.8.3 Échelle (Redimensionnement) . . . 66
2.8.4 Cisaillement (*Shear*) . . . 67
2.9 Résumé . . . 68
2.10 🤖 Utilisation de Gemini Notebook comme tuteur complémentaire . . . 68
2.11 Liste d'exercices . . . 69
Références du chapitre . . . 69
2.12 💻 **Partie Pratique avec Exercices de Programmation** . . . 69
🎯 Objectif de ce cahier . . . 70
2.12.1 EP02_01 ⊙ Réglage de la luminosité et du contraste . . . 70

2.12.2 EP02_02 🔬 Sous-échantillonnage spatial . . . 72
2.12.3 EP02_03 🎨 Quantification des niveaux de gris . . . 74
2.12.4 EP02_04 📐 Transformée de distance sur image binaire . . . 76
2.12.5 EP02_05 Translation d'image . . . 78
2.12.6 EP02_06 🔄 Rotation d'image . . . 79
2.12.7 EP02_07 🔍 Redimensionnement (Échelle) . . . 81
2.12.8 EP02_08 🔀 Cisaillement (Shear) . . . 85
2.12.9 EP02_09 🧩 Transformation Affine Générique . . . 86
2.12.10 EP02_10 🎯 Correction de perspective (homographie) . . . 88
2.12.11 EP02_11 🏆 Correction de Perspective (Homographie) sur une Image Réelle . . . 92

3 Opérations Spatiales : Intensité, Histogramme et Filtrage 97
3.1 Objetivos . . . 97
3.2 Opérations au Niveau d'Intensité . . . 97
3.2.1 Préparation de l'environnement pratique . . . 98
3.2.2 Opérations Arithmétiques . . . 99
3.2.3 Pondération Pondérée (*Alpha Blending*) . . . 100
3.2.4 Opérations Logiques et Masques Bit à Bit . . . 101
3.3 Histogramme d'images . . . 102
3.3.1 Égalisation d'histogramme . . . 103
3.3.2 Spécification d'histogramme . . . 107
3.4 Fondements spatiaux : Voisinage, convolution et *noyaux* . . . 108
3.4.1 Voisinage . . . 109
3.4.2 Traitement des bords . . . 109
3.4.3 Corrélation vs. Convolution . . . 110
3.4.4 Le Rôle du *Noyau* . . . 112
3.4.5 Exemple Numérique : Corrélation Pas à Pas . . . 115
3.5 Filtrage Spatial de Lissage . . . 116
3.5.1 Filtre Moyenneur (*Box Filter*) . . . 116
3.5.2 Filtre gaussien . . . 117
3.6 Filtrage Spatial de Rehaussement . . . 119
3.6.1 Laplacien . . . 119
3.6.2 Opérateur de Sobel . . . 123
3.6.3 Opérateur de Prewitt . . . 125
3.6.4 *Unsharp Masking* (USM) . . . 126
3.6.5 Détecteur de Canny . . . 128
3.7 Filtres d'Ordre : Filtre Médian . . . 129
3.7.1 Bruit Impulsif : Sel et Poivre . . . 129
3.7.2 Filtre Médian . . . 130
3.8 Application pratique : Prétraitement pour la segmentation . . . 133
3.9 Résumé . . . 134
3.10 🤖 Utilisation de Gemini Notebook comme tuteur complémentaire . . . 134
3.11 Liste d'exercices . . . 135
Références du chapitre . . . 136
3.12 💻 **Partie Pratique avec Exercices de Programmation** . . . 136
🎯 Objectif de ce cahier . . . 136
3.12.1 EP03_01 ➕ Addition Saturée de Constante . . . 137
3.12.2 EP03_02 🔀 Fusion *Alpha* de Deux Images . . . 139
3.12.3 EP03_03 🎭 Inversion d'image (négatif photographique) . . . 140
3.12.4 EP03_04 📊 Égalisation d'histogramme (L bits) . . . 142
3.12.5 EP03_05 🔲 Application du masque binaire ET . . . 143
3.12.6 EP03_06 Filtre de moyenne avec *kernel* N×N . . . 146
3.12.7 EP03_07 🔍 Opérateur Laplacien (w4) pour le Rehaussement des Contours . . . 149
3.12.8 EP03_08 🧭 Gradient de Sobel : Gx et Gy . . . 150

3.12.9 EP03_09 📡 Filtre Médian 3×3 . . . 152
3.12.10 EP03_10 ✨ *Unsharp Masking* (USM) . . . 155

4 Morphologie mathématique et segmentation d'images 159
4.1 Objectifs . . . 159
4.2 Seuilletage . . . 160
4.2.1 Image de pièces de monnaie . . . 161
4.2.2 Prétraitement pour Otsu . . . 161
4.2.3 Résultat : CLAHE comme meilleur prétraitement . . . 164
4.3 Morphologie mathématique . . . 165
4.3.1 Érosion et Dilatation . . . 165
4.3.2 Ouverture et Fermeture . . . 173
4.3.3 Opérateurs Géodésiques . . . 175
4.3.4 Reconstruction morphologique . . . 179
4.3.5 Remplissage des trous et suppression des bords . . . 181
4.3.6 Pipeline de Limpeza Binária com CLAHE . . . 184
4.3.7 Morphologie en niveaux de gris . . . 185
4.4 Segmentation d'Images : Fondements et Taxinomie . . . 188
4.4.1 Étiquetage . . . 189
4.4.2 Transformée de Distance . . . 193
4.4.3 Transformada de Distância Euclidiana em quatro passos . . . 196
4.4.4 Segmentation par *Watershed* . . . 198
4.5 Extraction de Composants et Descripteurs de Forme . . . 204
4.5.1 Rotation et statistiques des composants . . . 205
4.5.2 Descripteurs de forme . . . 206
4.5.3 Connexion avec la détection d'objets moderne . . . 208
4.6 Résumé . . . 210
4.7 🤖 Utilisation de Gemini Notebook comme tuteur complémentaire . . . 211
4.8 Liste d'exercices . . . 212
Références du chapitre . . . 212
4.9 💻 **Partie Pratique avec Exercices de Programmation** . . . 213
🎯 Objectif de ce carnet . . . 213
4.9.1 EP04_01 Seuillage global par seuil fixe . . . 214
4.9.2 EP04_02 📊 Seuillage automatique d'Otsu . . . 215
4.9.3 EP04_03 🌱 Dilatation binaire plane (mm.dil0) . . . 217
4.9.4 EP04_04 🪨 Érosion binaire plane (mm.ero0) . . . 219
4.9.5 EP04_05 🧹 Ouverture Morphologique (Suppression de Bruit) . . . 222
4.9.6 EP04_06 🧩 Fermeture Morphologique (Remplissage des Lacunes) . . . 225
4.9.7 EP04_07 Dilatation et Érosion Pondérées (mm.dil1 / mm.ero1) . . . 227
4.9.8 EP04_08 🌋 Gradient morphologique, Top-hat et Black-hat . . . 228
4.9.9 EP04_09 Transformée de distance et le « cœur » de l'objet . . . 231
4.9.10 EP04_10 🪙 Séparation des *blobs*, étiquetage et descripteurs . . . 233

5 Capítulo 5 — Segmentation d'images 237
5.1 5.1 Introduction . . . 237
5.2 5.2 Segmentation par seuillage . . . 237
5.3 5.3 Détection de contours . . . 237
5.4 5.4 Croissance de régions . . . 237
5.5 5.5 Exemples pratiques . . . 238
5.6 5.6 Résumé . . . 238

6 Transformées et Compression 239
6.1 Objectifs . . . 239
6.2 Configuration de l'environnement . . . 240

6.3 Transformada de Fourier Discreta 2D . . . 240
6.3.1 Simulateur : Reconstruire des signaux avec des sinusoïdes . . . 240
6.3.2 Interprétation du spectre de fréquences . . . 242
6.3.3 L'Expérience de la Grille : Construire une Image à partir d'un Unique Coefficient . . . 242
6.3.4 Définition mathématique . . . 244
6.3.5 Magnitude et Phase . . . 244
6.3.6 O que a Magnitude e a Fase carregam? . . . 246
6.3.7 Ce que portent la magnitude et la phase ? . . . 246
6.4 Théorème de convolution et stratégies de filtrage . . . 247
6.4.1 *Noyau* séparable vs. non séparable . . . 247
6.4.2 Analyse de l'efficacité computationnelle . . . 248
6.4.3 Discussion des résultats expérimentaux . . . 248
6.5 Filtres dans le Domaine Fréquentiel . . . 252
6.5.1 Filtres Passe-Bas . . . 253
6.5.2 Filtres Passe-Haut et Passe-Bande . . . 253
6.5.3 Suppression du bruit périodique . . . 257
📌 Synthèse — Filtres spectraux . . . 259
6.6 *Ondelettes* et Multirésolution . . . 259
6.6.1 La Limite de la Transformée de Fourier : localisation spatiale . . . 259
6.6.2 Transformée *Wavelet* Discrète 2D . . . 260
6.6.3 Familles d'*ondelettes* . . . 260
6.6.4 Analyse multirésolution avec la DWT 2D . . . 263
Synthèse — Fourier vs. *wavelets* : quand utiliser chaque approche ? . . . 270
6.7 Compression d'images . . . 270
6.7.1 Taxonomie des Redondances . . . 271
6.7.2 Transformée en Cosinus Discrète (DCT-II 2D) . . . 271
6.7.3 Les fonctions de base de la DCT . . . 271
6.7.4 Concentration d'Énergie et Reconstruction Progressive . . . 273
6.7.5 Le *Pipeline* de Compression JPEG . . . 275
6.7.6 Simulateur interactif : Quantification DCT . . . 278
6.8 Comparação de Formatos de Imagem . . . 279
6.8.1 Caractéristiques des Formats . . . 279
6.8.2 Métriques d'Évaluation de la Qualité . . . 279
6.8.3 Inspection Visuelle : Nature des Artefacts de Compression . . . 280
6.8.4 Évaluation Quantitative et Spatiale de la Compression . . . 281
Synthèse — Compression JPEG . . . 286
6.9 Application Pratique : Débruitage par Filtrage Hybride . . . 287
6.9.1 Analyse des performances et conclusion du chapitre . . . 287
6.10 Résumé du Chapitre . . . 289
Fundamentos Essenciais . . . 289
6.11 🤖 Utilisation de Gemini Notebook comme Tuteur Complémentaire . . . 291
6.12 Liste d'exercices . . . 291
Références du chapitre . . . 292
6.13 💻 **Partie pratique avec exercices de programmation** . . . 293
🎯 Objectif de ce cahier . . . 293
6.13.1 EP05_01 🟢 Filtre Passe-Bas Idéal par Distance dans le Spectre . . . 294
6.13.2 EP05_02 🟡 Filtre *Notch* : Suppression des pics périodiques . . . 295
6.13.3 EP05_03 🟠 Quantification DCT : la véritable source de compression . . . 297
6.13.4 EP05_04 🔴 Implémentation de la TFD 2D à partir de la définition . . . 300
6.13.5 EP05_05 🏆 *Pipeline* JPEG complet : DCT, quantification et reconstruction . . . 302

II Partie II — Vision par Ordinateur **305**

7 Analyse de documents et inspection industrielle **307**
7.1 Objectifs du chapitre . . . 307
7.2 Configuration de l'environnement . . . 308
7.3 Bases d'images pour l'expérimentation . . . 309
7.3.1 Images publiques avec `skimage.data` . . . 310
7.4 Normalisation du fond et égalisation locale de contraste . . . 312
7.5 Reconnaissance Optique de Caractères (OCR) . . . 314
7.6 Traduction automatique du texte reconnu . . . 316
7.7 Fondements de l'OMR et de l'Inspection Industrielle . . . 318
7.8 Projets Pratiques : Construction d'un *Pipeline* d'Analyse Documentaire . . . 319
7.8.1 Alignement Automatique de Documents (*Prétraitement OCR/OMR*) . . . 319
7.8.2 Algorithme de redressement d'inclinaison (*Deskew*) . . . 321
7.8.3 Modélisation Mathématique . . . 321
7.8.4 Limites pratiques . . . 324
7.8.5 Détection des bords et des contours . . . 325
7.8.6 Isollement, Segmentation et Décodage du *QRCode* . . . 327
7.8.7 Décodage de codes-barres . . . 331
7.9 Le MCTest comme étude de cas : du prototype au système en production . . . 334
7.9.1 Obtention et préparation du module . . . 334
7.9.2 Extraction de la zone de réponses . . . 335
7.9.3 Segmentation du *QRCode* . . . 337
7.9.4 Localisation des cadres de réponses . . . 339
7.9.5 Lecture automatique des réponses . . . 339
7.10 Inspection industrielle automatisée . . . 341
7.10.1 Références et *Datasets* publics . . . 342
7.10.2 Détection de défauts par analyse de texture . . . 343
7.11 Résumé . . . 346
Prochaines étapes . . . 347
7.12 🤖 Utilisation de Gemini Notebook comme tuteur complémentaire . . . 347
7.13 Liste d'exercices . . . 348
Références du chapitre . . . 349
7.14 💻 Partie pratique avec exercices de programmation . . . 349
Légende de difficulté . . . 350
🎯 Objectif de ce cahier . . . 350
7.14.1 EP06_01 🟢 Évaluation de Segmentation par IoU (*Intersection over Union*) . . . 351
7.14.2 EP06_02 🟢 Filtrage des marqueurs par circularité . . . 354
7.14.3 EP06_03 🟡 Classification des Marques sur les Feuilles de Réponses (OMR) . . . 355
7.14.4 EP06_04 🟡 Estimateur d'Inclinaison par Médiane Angulaire (*Deskew*) . . . 357
7.14.5 EP06_05 🟠 Normalisation du fond par division (correction de l'éclairage) . . . 359
7.14.6 EP06_06 🔴 Carte de Variance Locale pour la Détection de Texture . . . 361
7.14.7 EP06_07 🟣 *Pipeline* d'inspection industrielle : enregistrement par translation et soustraction . . . 363
7.14.8 EP06_08 ⚫ Segmentation et Décodage Réel de *QRCode* avec OpenCV . . . 366

8 Classification d'images et reconnaissance de formes **371**
8.1 Objectifs du chapitre . . . 371
8.2 Configuration de l'environnement . . . 371
8.3 Un Problème Concret : Classification des Fruits . . . 373
8.4 Aperçu du Chapitre : Le *Pipeline* Classique de Classification d'Images . . . 375
8.5 Fondements de la reconnaissance de formes . . . 375
8.6 Extraction de descripteurs classiques . . . 376
8.6.1 *Local Binary Patterns* (LBP) . . . 377

8.6.2 *Histogram of Oriented Gradients* (HOG) . . . 379
8.6.3 L'Impact de l'Échelle et la Normalisation des Caractéristiques . . . 381
8.7 📌 Carte conceptuelle . . . 383
8.8 Descripteurs en pratique . . . 383
8.9 Um Problema Concreto : Simulation de Descripteurs de Fruits . . . 388
8.10 Classificateur k-NN : Comment il fonctionne en interne . . . 391
8.10.1 Métrique de distance . . . 392
8.10.2 Règle de décision . . . 392
8.10.3 Le rôle du paramètre k . . . 394
8.11 Projet Pratique 1 : Classification de Chiffres Manuscrits avec k-NN . . . 395
8.11.1 Classification avec des vecteurs d'intensité . . . 396
8.11.2 Classification avec des descripteurs HOG . . . 398
8.12 Évaluation des Classifieurs . . . 399
8.12.1 Choix de k par Validation Croisée . . . 399
8.12.2 Le compromis entre biais et variance : Diagnostic de *surajustement* et de *sous-apprentissage* . . . 400
8.13 Projet Pratique 2 : Comparaison de Descripteurs pour la Classification de Textures . . . 402
8.13.1 Pipeline d'extraction de caractéristiques et évaluation comparative . . . 404
8.13.2 Diagnostic fin du classificateur : précision, rappel et F1-score . . . 407
8.14 Limites des descripteurs artisanaux . . . 409
8.15 Résumé . . . 410
8.16 🤖 Utilisation de Gemini Notebook comme tuteur . . . 410
8.17 Liste d'exercices . . . 410
Références du chapitre . . . 411
8.18 💻 **Partie Pratique avec des Exercices de Programmation** . . . 412
🎯 Objectif de ce Carnet . . . 412
Téléchargement . . . 412
Résumé des méthodes de `morph.py` (Chap. 7) . . . 413
8.18.1 EP07_01 🟢 Classifieur k-NN Pas à Pas . . . 413
8.18.2 EP07_02 🟡 Normalisation *Z-score* et Robustesse du k-NN à des Échelles Distinctes . 415
8.18.3 EP07_03 🟡 Évaluation par Matrice de Confusion . . . 417
8.18.4 EP07_04 🟠 Codage Manuel du Descripteur LBP . . . 420
8.18.5 EP07_05 🔴 Histogramme des orientations d'une cellule HOG . . . 421
8.18.6 EP07_06 🟣 *Pipeline* complet : Descripteurs + k-NN + Évaluation multi-classe . . . 424
8.18.7 EP07_07 ⚫ Classification réelle d'une mosaïque de textures via LBP + k-NN . . . 428

9 Comprendre les scènes : Mise en correspondance de caractéristiques, détection et segmentation 435
9.1 Objectifs du chapitre . . . 435
9.2 Configuration de l'Environnement . . . 436
9.3 Detecteurs et Descripteurs Locaux de Caractéristiques . . . 437
9.3.1 L'algorithme ORB (*Oriented FAST and Rotated BRIEF*) . . . 437
9.3.2 Exploration du simulateur ORB . . . 438
9.4 Correspondance de caractéristiques et homographie avec ORB . . . 440
9.4.1 Détection et mise en correspondance des caractéristiques avec ORB . . . 441
9.5 Modélisation mathématique : homographie et RANSAC . . . 443
9.5.1 Homographie . . . 443
9.5.2 RANSAC . . . 443
9.5.3 Estimation de l'homographie avec RANSAC . . . 445
9.5.4 Enregistrement de l'image . . . 446
9.6 Détection d'objets : *Haar Cascade* (Viola-Jones) . . . 448
9.7 Détection d'Objets : Boîtes Englobantes, IoU et NMS . . . 450
9.7.1 *Intersection over Union* (IoU) . . . 451
9.7.2 Suppression Non-Maximale (NMS) . . . 451

9.8 Segmentation visuelle : sémantique, par instances et panoptique . . . 453
9.8.1 Segmentation sémantique par seuillage . . . 453
9.8.2 Segmentation d'instances par étiquetage . . . 453
9.8.3 Alternative : Transformée de distance + *Watershed* . . . 455
9.8.4 Mesure et Extraction des Attributs avec `mm.measure` . . . 457
9.8.5 Exportation des annotations : de `mm.measure` au format YOLO . . . 459
9.8.6 Validação de Anotações por IoU : `mm.verifyBoundBox` . . . 460
9.8.7 Visualisation des annotations sauvegardées : `mm.showBoundBox` . . . 461
9.9 Aperçu des modèles modernes . . . 462
9.10 Limites des Approches Classiques et Motivation pour le *Deep Learning* . . . 463
9.11 Résumé . . . 463
Étapes suivantes . . . 464
9.12 🤖 Utilisation du Gemini Notebook comme Aide à l'Étude . . . 464
9.13 Liste d'exercices . . . 464
Références du chapitre . . . 465
9.14 💻 **Partie Pratique avec Exercices de Programmation** . . . 466
🎯 Objectif de ce Carnet . . . 466
Téléchargement . . . 466
Résumé des méthodes de `morph.py` (Chap. 8) . . . 467
9.14.1 EP08_01 🟢 Distance de Hamming et correspondance de descripteurs binaires . . . 467
9.14.2 EP08_02 🟢 Homographie et RANSAC : Le vote par *inliers* . . . 469
9.14.3 EP08_03 🟢 Image intégrale : sommes rectangulaires en temps constant . . . 471
9.14.4 EP08_04 🟢 IoU et Suppression des Non-Maximums (NMS) . . . 473
9.14.5 EP08_05 🟡 Étiquetage des Composantes Connexes : Segmentation d'Instances . . . 475
9.14.6 EP08_06 🟡 *Bounding Boxes*, Centroïdes et Propriétés des Instances avec `mm.measure` 478
9.14.7 EP08_07 🟡 Suppression du Bruit Sel et Poivre et Mesure d'Objets . . . 479
9.14.8 EP08_08 🟡 Image en Niveaux de Gris et Seuillage Dynamique . . . 481
9.14.9 EP08_09 🟠 Gradient d'éclairage et seuillage adaptatif . . . 484
9.14.10 EP08_10 🔴 Contraste Faible et Séparation d'Objets Thèmes (*Watershed* / Distance) 486
9.14.11 EP08_11 🔴 Classification et Validation d'Objets avec Gabarit de *Bounding Box* . . . 488
9.14.12 EP08_12 🔴 Segmentation d'instances sur image réelle : objets géométriques . . . 491

10 Apprentissage profond pour la vision par ordinateur **495**
10.1 Objectifs du chapitre . . . 495
10.2 Aperçu : Classification, Détection et Segmentation . . . 496
10.3 Configuration de l'environnement . . . 497
10.4 Fondements de l'Apprentissage Profond pour la VC . . . 499
10.4.1 Da Convolução Fixa à Convolução Aprendida . . . 499
10.4.2 De la convolution fixe à la convolution apprise . . . 499
10.4.3 Neurone Artificiel . . . 500
10.4.4 Couche convolutive . . . 502
10.4.5 Fonction d'activation . . . 502
10.4.6 *Pooling* . . . 504
10.4.7 Entraînement des réseaux de neurones : comment les CNN apprennent . . . 504
10.4.8 Architecture d'une CNN . . . 507
10.4.9 Comment le Gradient Ajuste les *Kernels* de la Convolution . . . 509
10.5 Applications Pratiques en CV . . . 514
10.5.1 Classification d'images avec les CNN . . . 514
10.5.2 Détection d'objets . . . 545
10.5.3 Segmentation d'objets . . . 556
10.5.4 Ingénierie des données pour la vision par ordinateur (*Roboflow*) . . . 566
10.5.5 Applications géométriques et intégrées . . . 571
10.6 Résumé . . . 576
10.7 🤖 Utilisation de Gemini Notebook comme tuteur complémentaire . . . 577

10.8 Liste d'exercices . . . 577
10.9 Clôture de la Partie II . . . 579
Références du chapitre . . . 579
10.10 💻 **Partie Pratique avec des Exercices de Programmation** . . . 580
🎯 Objectif de ce Carnet . . . 581
10.10.1 EP09_01 🟢 Convolution 2D Manuelle (*Forward* d'une Couche Apprise) . . . 581
10.10.2 EP09_02 🟢 *Pooling* manuel (Maximum et moyenne) . . . 585
10.10.3 EP09_03 🟡 Comptage des paramètres entraînables d'un CNN . . . 588
10.10.4 EP09_04 🟡 Intersection sur Union (IoU) et Suppression des Non-Maximums (NMS) 591
10.10.5 EP09_05 🟠 Évaluation de segmentation : IoU et Dice pixel par pixel . . . 594
10.10.6 EP09_06 🔴 *Pipeline* intégré : de la détection à la mesure du monde réel . . . 599

Références **603**

Annexes **605**

A À propos de MCTest **605**
A.1 Installation de MCTest . . . 605
A.2 Créer un examen dans MCTest . . . 605
A.3 Créer une question paramétrique . . . 606
A.4 Exemple réel : une question du Simulado 4 . . . 608
A.5 Remarques finales . . . 610

B À propos de Moodle (VPL) **611**
B.1 Configurer une activité VPL . . . 611
B.2 Publier les EPs comme activités VPL . . . 611
B.3 Remarques finales . . . 612

C Utilisation du SEB lors des simulacres d'examen bimensuels et des évaluations **613**
C.1 Configurer l'application dans *SEB Tool* . . . 613
C.2 Associer la clé SEB à l'activité VPL dans Moodle . . . 614
C.3 Restriction par adresse IP (facultatif) . . . 614
C.4 Remarques finales . . . 614

D Génération de matériel pédagogique avec *Gemini Notebook* **615**
D.1 Vidéos conceptuelles et de résolution des EPs . . . 615
D.2 *Slides* d'appui . . . 615
D.3 Matériel principal et remarques finales . . . 615

E Utilisation de l'IA dans l'évaluation des étudiants : le projet vpl-ai-feedback **617**
E.1 Contexte d'utilisation . . . 617
E.2 Architecture du projet . . . 618
E.3 Le *prompt* universel et la grille en deux étapes . . . 619
E.4 Deux couches supplémentaires de preuves . . . 620
E.5 Déroulement de l'exécution . . . 621
E.6 Exemple illustratif : grille pour un examen à trois questions . . . 621
E.7 Envoi du *feedback* par e-mail . . . 624
E.8 Risques, limites éthiques et responsabilité de l'enseignant . . . 625
E.9 Remarques finales . . . 626

Liste des figures

1 Bouton cliquable **Exécuter Colab**, disponible au début des parties **Théorique** et **Pratique** de chaque chapitre, permettant l'exécution interactive des exemples et des exercices. Dans la version PDF, il existe un *lien* HTML direct entre l'image du simulateur et sa légende. 5

1.1 Diagramme relationnel détaillant les distinctions fondamentales, les synergies et les interconnexions entre le TNI et la VO dans le contexte des systèmes basés sur l'image. 4
1.2 Représentation du flux séquentiel de traitement : la sortie de chaque niveau devient l'entrée du niveau suivant. 6
1.3 Flux détaillé du PDI : de l'acquisition sensorielle à l'extraction d'attributs et à la reconnaissance automatisée, incluant le traitement couleur et la compression. 6
1.4 Représentation du spectre visible et de sa position par rapport aux autres rayonnements électromagnétiques, mettant en évidence la variation des longueurs d'onde de 380 nm à 750 nm. 7
1.5 (A) Spectre électromagnétique complet en échelle logarithmique, avec mise en évidence de la bande visible. (B) Détail de la lumière visible (380-750 nm) et de ses couleurs. (C) Décomposition de la lumière blanche par le prisme : la longueur d'onde la plus courte subit une réfraction plus grande, séparant UV, visible et infrarouge. 8
1.6 Exemples d'images synthétiques représentées matriciellement. 11
1.7 Mandrill (Mandrillus sphinx) en milieu naturel. Crédit : Julien Renoult (CC BY 4.0). 12
1.8 Traitement de base des images : (a) image originale, (b) image en niveaux de gris, (c) image binarisée par seuil (T=128). 14
1.9 Comparaison entre seuillage manuel (T=128) et automatique (Otsu) sur l'image en niveaux de gris. 15
1.10 Acesso a um pixel específico (r, c) e a representação de sua vizinhança na imagem. 17
1.11 Simulateur EP01_01 : Distances Euclidienne, City-block et Échiquier 24
1.12 Simulateur EP01_02 : Performance Prédictive — Métriques de ML 31
1.13 Simulateur EP01_03 : Mean Average Precision (mAP) et courbe P-S 35
1.14 Simulateur EP01_04 : Statistiques de pixels dans une matrice discrète 5x5 36
1.15 Simulateur EP01_05 : Transformation de Négatif d'Image (Inversion d'Intensité) 38
1.16 Simulador EP01_06 : Conversion RVB en niveaux de gris (pondération perceptuelle UIT-R BT.601) 39
1.17 Simulateur EP01_07 : Seuillage global interactif (binarisation p > T) 41
1.18 Simulateur EP01_08 : Remappage par Plage Conditionnelle (Luminosité et Contraste par Seuil) 43
1.19 Simulateur EP01_09 : Générateur de motif en damier (logique de parité (i + j) % 2) 44
1.20 Simulateur EP01_10 : Structure du fichier PGM (En-tête ASCII P2 et mappage vers un tuple Python) 46
1.21 Simulateur EP01_11 : Filtre de Maximum Local 1D (Voisinage 1x3 avec Condition de Bord) 48

2.1 Comparaison didactique entre le système visuel biologique et le système électronique : en haut, l'anatomie de l'œil humain mettant en évidence la rétine et les photorécepteurs (cônes et bâtonnets) ; en bas, la structure d'un appareil photo numérique détaillant le capteur CMOS, la matrice de filtres de Bayer (RGGB) et le processus de quantification numérique réalisé par le convertisseur analogique-numérique (ADC). 50
2.2 Recueil de défis perceptifs : (en haut à gauche) Vase de Rubin — ambiguïté figure-fond ; (en haut au centre) Éléphant de Shepard — incongruence géométrique ; (en haut à droite) Ombre d'Adelson — constance de la luminosité fondée sur le contexte ; (en bas à gauche) Grille de points — inhibition latérale ; (en bas à droite) Escalier de Schröder. 51

2.3 Imagem RGB (3 canais, $M \times N \times 3$) e versão RGBA (4 canais, $M \times N \times 4$) com transparência alfa gradual da esquerda para a direita. Imagem Mandrill — USC SIPI Image Database (domínio público). . . . 55
2.4 Effet du sous-échantillonnage. Les titres affichent les dimensions (W x H) et la taille de la matrice en mémoire (Ko). . . . 57
2.5 Effet de la réduction de la profondeur de bits. Les titres affichent la quantité de bits, les niveaux et la taille en mémoire (Ko). . . . 58
2.6 Illustration des voisinages 4 dans une matrice 3x3. Au centre (1,1), le pixel d'intérêt. . . . 59
2.7 Zone de Protection Environnementale Quilombos do Médio Ribeira - Barbudo-rajado (Malacoptila striata). Crédit : Thomas Fuhrmann (CC BY-SA 4.0). . . . 63
2.8 Exemple de translation de l'image de l'oiseau avec des déplacements (50,50) et (100,50). . . . 65
2.9 Exemple de rotation de l'image de l'oiseau à 30° et 45° en utilisant l'interpolation bilinéaire. . . 66
2.10 Comparaison de l'interpolation avec zoom sur le détail de l'œil (recadrage 60×60, agrandi 4×). Notez l'effet de blocs dans le plus proche voisin vs. l'adoucissement en bilinéaire. . . . 67
2.11 Exemple de cisaillement de l'image de l'oiseau : horizontal (shx=0.3), vertical (shy=0.3) et combiné (shx=0.2, shy=0.2). . . . 68
2.12 Simulateur EP02_01 : Ajustement de la Luminosité et du Contraste Linéaire (p' = p +) . 72
2.13 Simulateur EP02_02: Sous-échantillonnage spatial (réduction de résolution par saut f) 74
2.14 Simulateur EP02_03 : Quantification et Profondeur de Bits (Réduction du Nombre de Niveaux de Gris) . . . 76
2.15 Simulateur EP02_04 : Transformée de distance en image binaire (Chessboard, City-block et Euclidienne) . . . 78
2.16 Simulateur EP02_05 : Translation géométrique d'image (Déplacement tx et ty avec remplissage de bordure) . . . 80
2.17 Simulateur EP02_06 : Rotation d'image autour de l'origine par un angle . . . 82
2.18 Simulateur EP02_07 : Redimensionnement spatial et interpolation (Nearest Neighbor vs Bilinéaire) . . . 84
2.19 Simulateur EP02_08 : Transformation géométrique de cisaillement 2D (Cisaillement horizontal et vertical) . . . 87
2.20 Simulador EP02_09 : Transformação Afim 2D (Matriz 2×3) . . . 89
2.21 Simulateur EP02_10 : Correction de perspective (Transformation d'homographie 3×3) 91
2.22 Aquisição da imagem de um Sudoku à esquerda. À direita, conversão para tons de cinza e redimensionamento. Crédito: Héctor Rodríguez de Guardamar, Espanha (CC BY 2.0). . . . 93
2.23 Correction de perspective : vue originale et vue frontale redressée. . . . 95
2.24 Simulateur EP02_11 : Correction de perspective du Sudoku (Homographie 3×3 avec rééchantillonnage bilinéaire) . . . 95

3.1 *Mandrill* (*Mandrillus sphinx*) photographié dans son environnement naturel en Afrique du Sud. Crédit : Carlos Guilherme Rodrigues (CC BY-SA 3.0). . . . 99
3.2 Opérations arithmétiques saturées : addition de constante (éclaircissement) et soustraction de constante (assombrissement avec saturation à 0). . . . 100
3.3 Portrait d'un léopard (*Panthera pardus*) dans son environnement naturel. Crédit : C. Brück (CC BY-SA 4.0). . . . 101
3.4 *Mélange alpha* entre des découpes alignées du mandrill et du léopard (Figure 3.3) pour différentes valeurs de . À =1, on ne voit que le mandrill ; à =0, seulement le léopard ; les valeurs intermédiaires fusionnent les regards des deux images proportionnellement. . . . 101
3.5 Opérations logiques bit à bit avec masque circulaire : AND (isolement de la ROI), OR (illumination de la ROI) et NOT (négatif). . . . 102
3.6 Histogrammes de l'image originale, d'une version assombrie (−80) et d'une éclaircie (+80). La soustraction/addition sature à 0 et 255. . . . 103
3.7 Égalisation d'histogramme sur une image 5×5 avec 3 bits (L=8) : image originale avec des tons concentrés sur {2,3,4}, image égalisée avec des tons redistribués sur {1,5,7}, et les histogrammes respectifs montrant l'étalement des fréquences. . . . 106

3.8 Égalisation d'histogramme : mm.equalize (global via CDF) vs. CLAHE (adaptative avec limitation de contraste). Les histogrammes révèlent l'étalement progressif des intensités. . . . 107
3.9 Spécification d'histogramme : mandrill mappé sur le profil tonal du léopard. La CDF de sortie approxime la CDF de référence. . . . 108
3.10 Corrélation avec *noyau* de moyenne 3×3 : version didactique (mm.conv0) vs. vectorisée (mm.conv via cv2.filter2D). La différence maximale de 39 se produit sur les bords. . . . 114
3.11 *Patch* 5×5 extraído da imagem do leopardo (posição [250:255, 250:255]). A janela amarela destaca a vizinhança 3×3 centrada no pixel [1,1] onde a correlação será calculada. . . . 115
3.12 Filtre de moyenne avec des *kernels* de taille croissante (3×3, 7×7, 15×15). Le flou des bords augmente avec la taille du *kernel*. . . . 116
3.13 *Noyau* gaussien 5×5 (=1) : poids normalisés, plus grands au centre et décroissant radialement. La grille facilite la lecture de chaque coefficient. . . . 117
3.14 Comparaison entre filtre de moyenne et gaussien (*noyau* 9×9, =0). Le gaussien préserve mieux les bords, visible dans le détail du visage. . . . 119
3.15 Simulateur : Filtrage spatial d'accentuation 1D (Unsharp Masking et High-Boost) . . . 120
3.16 *Noyau* Laplacien w4 appliqué au *patch* 5×5 : la fenêtre jaune met en évidence le voisinage 3×3 où l'opérateur de dérivée seconde est calculé. . . . 122
3.17 Laplacien appliqué à l'image du léopard : réponse brute (bords) avec w4 et w8, et images rehaussées par la soustraction du Laplacien. w8 est plus sensible aux diagonales. . . . 123
3.18 Opérateur de Sobel sur l'image du léopard : Gx détecte les bords verticaux, Gy détecte les bords horizontaux, et la magnitude | f| combine les deux, révélant tous les bords indépendamment de la direction. . . . 125
3.19 Opérateur de Prewitt : Gx, Gy et magnitude — comparable à Sobel, mais sans pondération gaussienne perpendiculaire. . . . 125
3.20 Rehaussement de netteté par *Unsharp Masking* sur l'image du léopard : l'image lissée est soustraite de l'originale pour générer le masque haute fréquence, qui est ensuite réintroduit pour accentuer les contours et les détails. . . . 127
3.21 *Démasquage non net* sur l'image du léopard avec =1 et k de 0,5 à 8,0. Pour k>2 des halos apparaissent sur les bords et le bruit de fond ressort. . . . 128
3.22 Détecteur de Canny avec différentes paires de seuils : des seuils bas capturent plus de bords (y compris le bruit) ; des seuils élevés ne conservent que les bords les plus forts. . . . 129
3.23 Bruit sel et poivre avec des densités croissantes (2 %, 5 %, 10 %). Le paramètre prob indique la fraction totale de pixels corrompus, moitié sel (255) et moitié poivre (0). . . . 130
3.24 Comparaison de filtres pour la suppression du bruit sel et poivre (10 %) : Gaussien, Moyenne, Médiane, Bilatéral et Morphologique. Ligne supérieure : images complètes ; ligne inférieure : recadrage de la région d'intérêt. . . . 133
3.25 *Pipeline* de pré-processamento : CLAHE → Gaussiano → Canny. Cada etapa prepara a imagem para a seguinte, resultando em bordas limpas e bem definidas. . . . 134
3.26 Simulateur EP03_01 : Addition saturée de constante (p' = clip(p + k)) . . . 138
3.27 Simulateur EP03_02 : Mélange Alpha de Deux Images (g = · f1 + (1−) · f2) . . . 140
3.28 Simulador EP03_03 : Inversion d'image — Négatif photographique (p' = 255 − p) . . . 142
3.29 Simulateur EP03_04 : Égalisation d'histogramme (Niveaux L = 2^B) . . . 144
3.30 Simulateur EP03_05: Application de masque AND binaire . . . 146
3.31 Simulateur EP03_06: Filtre de Moyenne avec Noyau N×N . . . 148
3.32 Simulateur EP03_07 : Opérateur Laplacien (w4) pour le rehaussement des contours . . . 151
3.33 Simulateur EP03_08 : Gradient de Sobel (Gx et Gy) . . . 153
3.34 Simulateur EP03_09 : Filtre de la Médiane 3×3 . . . 155
3.35 Simulateur EP03_10 : *Unsharp Masking* (USM) . . . 157

4.1 Image avec des pièces de plusieurs types. Crédit : GAZI.MD.AHAD (CC BY-SA 4.0). . . . 161
4.2 Comparaison des prétraitements : image | histogramme+T* | ²B(T) | Otsu. La meilleure séparation bimodale indique le seuil le plus fiable. . . . 164
4.3 Élément structurant en forme de croix (B_{cruz}) utilisé pour la connectivité-4. . . . 166

4.4 Érosion et dilatation sur une image binaire 10×10 avec un élément structurant « L » asymétrique 3×3. Validation de la dualité érosion–dilatation. 172
4.5 Simulateur : Érosion Morphologique (A B_L) 172
4.6 Ouverture, fermeture, composition et filtre séquentiel alterné appliqués à la binarisation Otsu des pièces (prétraitées par rehaussement de contraste). Élément structurant : disque 13×13. . 174
4.7 Simulateur interactif de dilatation géodésique : le marqueur f (vert) se propage pas à pas le long des chemins libres du masque g, sans traverser les murs. 176
4.8 Résolution de labyrinthe via Opérations Complémentaires : en inversant le masque et le marqueur au début du *pipeline*, le flux est résolu directement dans le domaine complémentaire via mm.cero et mm.suprec, éliminant les ré-inversions redondantes. 179
4.9 *Pipeline* de Reconstruction morphologique par Dilatation conditionnée : le masque contient deux objets, le marqueur isole uniquement le noyau de l'objet principal, et les itérations reconstruisent sa forme exacte jusqu'à la convergence. 181
4.10 *Pipeline* de preenchimento de buracos (*clohole*) : o marcador vem da borda da imagem, restrito ao complemento f^c. A dilatação geodésica reconstrói o fundo externo ; após a complementação, os buracos internos ficam preenchidos. 183
4.11 *Pipeline* de suppression des structures de bordure (*edgeoff*) : le marqueur capture les racines connectées aux extrémités, la reconstruction délimite ces éléments et la soustraction ne préserve que les objets totalement internes. 184
4.12 *Pipeline* complet de segmentation avec CLAHE : Otsu → ouverture (r=9) → clohole → ouverture (r=33) → edgeoff. 185
4.13 Simulateur interactif avancé de morphologie avec élément structurant modifiable et profil 1D. 186
4.14 Morphologie en niveaux de gris et leurs histogrammes respectifs. L'érosion réduit les intensités locales, la dilatation les amplifie, le gradient met en évidence les bords, et les opérateurs Top-hat et Black-hat révèlent les détails locaux clairs et sombres. Élément structurant : disque de diamètre 19. 188
4.15 Algorithme d'étiquetage par *flood-fill* avec pile. 191
4.16 Simulateur interactif d'étiquetage de composantes connexes (*connected component labeling*) : visualisation de l'expansion flood-fill, connectivité-4 et connectivité-8. 192
4.17 Effet de la connectivité sur l'étiquetage : l'image binaire 10×10 contient des pixels adjacents en diagonale. Avec la connectivité-4, ces pixels forment des composantes distinctes ; avec la connectivité-8, ils fusionnent en une seule composante. 192
4.18 Algorithme de la Transformée de Distance. 194
4.19 Simulateur interactif de la Transformée de Distance (TD) itérative via érosion en niveaux de gris. Les pixels hors de l'image prennent la valeur maximale (144), propageant les coûts depuis le fond interne. 195
4.20 Transformée de distance sur image binaire 10×10. À gauche : originale (*foreground* = 255). Au centre : mm.dist1 itérative (érosions avec croix). À droite : mm.dist (L2). 196
4.21 Simulateur interactif 2D (4x4) avec synchronisation stricte des files de propagation horizontale (b) pour obtenir la convergence exacte décrite dans l'article. 197
4.22 Plus court chemin géodésique dans un labyrinthe. Les distances géodésiques sont calculées à partir de l'entrée et de la sortie en utilisant mm.gdist. Les pixels dont la somme des deux distances est égale à la distance minimale entre les marqueurs appartiennent à un chemin optimal. 198
4.23 Algorithme didactique de *Watershed* par croissance de régions limitée par masque. 200
4.24 Simulateur interactif de l'algorithme *Watershed* par propagation morphologique. Dessinez le masque, positionnez les marqueurs et ajustez l'élément structurant pour observer l'inondation. Lorsque les bassins se rencontrent simultanément, l'égalité est résolue en supposant l'une des régions de manière aléatoire. 201
4.25 *Pipeline watershed* délimité par masque sur image binaire 20×20. 202
4.26 *Pipeline Watershed* pour séparation de pièces superposées : de la masque binaire dilatée jusqu'aux contours finaux annotés sur l'image originale. 204

4.27 Composantes connexes extraites après le *pipeline* CLAHE → Otsu → nettoyage morphologique. Chaque objet est coloré d'une couleur distincte et annoté avec son aire en pixels. 206
4.28 Contours extraits avec *findContours*. Chaque pièce est annotée avec sa circularité — des valeurs proches de 1 confirment une forme circulaire. 208
4.29 *Boîtes englobantes* dérivées des composantes connexes superposées à l'image originale. Les annotations sont exportées au format YOLO (*classe cx cy w h*), avec des coordonnées normalisées pour l'intervalle [0,1]. 209
4.30 Simulateur EP04_01 : Seuillage global par seuil fixe (p' = (p > T) ? 255 : 0) 215
4.31 Simulateur EP04_02 : Seuillage automatique d'Otsu (T* = argmax ²_B(T)) 217
4.32 Simulateur EP04_03 : Dilatation binaire plane (g = f B) 220
4.33 Simulateur EP04_04: Érosion Binaire Plane (g = f B) 222
4.34 Simulateur EP04_05 : Ouverture morphologique (g = (f B) B) 224
4.35 Simulateur EP04_06: Fermeture morphologique (g = (f B) B) 226
4.36 Simulateur EP04_07 : Dilatation et Érosion avec Poids (mm.dil1 / mm.ero1) 229
4.37 Simulateur EP04_08: Gradient morphologique, Top-hat et Black-hat 231
4.38 Simulateur EP04_09: Transformée de Distance (Couches d'Érosion) 233
4.39 Simulateur EP04_10: Séparation des Blobs, Étiquetage et Descripteurs 235

6.1 Décomposition de Fourier 1D : une onde carrée (ligne pointillée) est approximée par la somme des premières sinusoïdes (lignes colorées). Plus il y a de termes, meilleure est l'approximation. 240
6.2 Simulateur interactif de la décomposition de Fourier 1D : visualisation de la somme de sinusoïdes avec différentes fréquences, amplitudes et phases. Ajoutez des termes et observez la convergence vers des formes d'onde arbitraires. 241
6.3 Toute fréquence dans le spectre (point isolé) correspond à une onde sinusoïdale 2D tournée dans le domaine spatial. 243
6.4 Simulateur interactif de la synthèse de Fourier 2D. Modifiez la position horizontale (u) et verticale (v) du coefficient dans le spectre de fréquences centré et observez comment la distance par rapport au centre dicte la fréquence spatiale (épaisseur) et l'angle dicte l'orientation de l'onde sinusoïdale générée. 243
6.5 Diagramme conceptuel du spectre de Fourier 2D centré. 245
6.6 Expérience de changement de phase : Image A (pièces) et Image B (motif géométrique) reconstruites avec magnitudes et phases échangées. Le résultat montre que la structure visuelle est **beaucoup plus sensible à la phase** qu'à la magnitude : lorsque la phase de B est conservée, l'image résultante préserve l'organisation spatiale de B, même avec la magnitude de A. La magnitude, quant à elle, influence principalement le contraste et la texture. Observez que la qualité de la reconstruction n'est pas parfaite — des artefacts visibles sont présents —, soulignant l'interdépendance entre phase et magnitude pour une représentation fidèle de l'image. 247
6.7 Comparaison d'efficacité : Convolution non séparable (spatiale 2D), séparable (spatiale 1D) et via FFT. 249
6.8 Sans *padding*, un décalage sévère fait fuir l'image vers le côté opposé (convolution circulaire). 250
6.9 Vérification du Théorème de Convolution : la différence pixel à pixel entre la convolution spatiale (cv2.filter2D) et la multiplication en fréquence (FFT) est numériquement nulle — confirmant l'équivalence théorique. 251
6.10 La Dualité Dangereuse : La coupure brutale en Fréquence (Cylindre) se transforme obligatoirement en Sinc spatiale. Ses ondulations provoquent le *ringing* fantôme sur les bords de l'image. 252
6.11 Filtros passe-bas. 253
6.12 Simulateur interactif de filtres dans le domaine fréquentiel. 254
6.13 Comparaison entre filtres passe-bas : idéal (D =30), gaussien (D =30) et Butterworth (D =30, n=2). Profils de H(u,v) le long d'une ligne centrale et images filtrées correspondantes. 256
6.14 Filtre passe-haut gaussien. (a) Original ; (b) Filtre passe-haut (D =30) - les bords des pièces et le fond texturé sont mis en valeur. 257

6.15 Suppression du bruit périodique via filtre *notch* dans le domaine fréquentiel : (a) image avec bruit sinusoïdal, (b) spectre montrant les pics de bruit, (c) masque *notch* centré sur les pics, (d) image restaurée. 258
6.16 La transformée de Fourier globale est aveugle à la position. Les spectres ne disent pas où se trouvent les bords. 260
6.17 Fonctions de la *wavelet* ( ). Notez comme elles décroissent rapidement vers zéro (support compact), contrairement aux sinusoïdes infinies de Fourier. 262
6.18 Schéma de la décomposition *wavelet* 2D en deux niveaux. 262
6.19 Simulation de la décomposition *wavelet* 2D. 264
6.20 Décomposition *wavelet* 2D à 2 niveaux avec *wavelet* de Haar : sous-bandes LL, LH, HL, HH à chaque niveau. Les sous-bandes de détail révèlent des structures orientées à différentes échelles en utilisant un motif synthétique. 266
6.21 Comparaison entre famílias de *ondelettes* : Haar, db4, sym4 et bior2.2. Sous-bande LL (approximation) et HH (diagonale) pour chaque choix, illustrant le compromis entre compactage et douceur sur la base du motif synthétique. 267
6.22 Reconstruction *wavelet* avec seuillage des coefficients (*hard thresholding*) : à mesure que le seuil augmente, plus de détails sont mis à zéro, produisant des images progressivement plus lisses. La métrique PSNR quantifie la perte de qualité sur le motif synthétique. 269
6.23 L'Alphabet visuel du JPEG : Les 64 fonctions de base de la DCT-II. Le coefficient DC se trouve en haut à gauche (lisse). En descendant et en allant vers la droite, l'oscillation spatiale augmente drastiquement. 273
6.24 DCT 2D em bloco 8×8: coeficientes e reconstrução progressiva. 275
6.25 *Pipeline* JPEG simplifié appliqué à l'image classique du *Cameraman* : DCT en blocs 8×8, quantification avec différents facteurs de qualité et reconstruction via IDCT. Les artefacts de bloc (*blocking artifacts*) deviennent visuellement évidents avec des facteurs de qualité réduits ($Q = 10$ et $Q = 25$). 278
6.26 Simulateur interactif de compression DCT-JPEG : ajustez le facteur de qualité et visualisez en temps réel les coefficients nuls, le bloc reconstruit et l'erreur de quantification. 279
6.27 Analyse comparative des artefacts de compression sous facteur de qualité réduit ($Q = 10$). À gauche, on observe l'artefact de bloc caractéristique de la discrétisation par DCT dans le JPEG. À droite, on met en évidence l'effet d'atténuation et d'adoucissement des bords intrinsèque au format WebP. 281
6.28 Courbe taux-distorsion : PSNR vs taille de fichier pour JPEG, WebP et PNG appliquée à l'image du *Cameraman*. 283
6.29 Analyse spatiale de la dégradation : images reconstruites et cartes d'erreur absolue normalisées associées pour différents facteurs de qualité JPEG. 285
6.30 *Pipeline* complet de suppression du bruit mixte : (1) ajout de bruit gaussien et périodique ; (2) identification des pics d'interférence dans le spectre de fréquences ; (3) application d'un masque *notch* avec atténuation gaussienne douce ; (4) post-traitement via filtre bilatéral pour élimination du bruit stochastique résiduel. 289
6.31 Carte conceptuelle des transformations et propriétés dans le domaine fréquentiel. 290
6.32 Simulateur EP05_01 : Filtre passe-bas idéal dans le spectre 296
6.33 Simulateur EP05_02: Filtre Notch 298
6.34 Simulateur EP05_03 : Quantification DCT (*round-trip*) 299
6.35 Simulateur EP05_04: DFT 2D manuel 301
6.36 Simulateur EP05_05 : *Pipeline* JPEG complet par bloc 304

7.1 Échantillon d'images publiques de *skimage.data*, groupées par domaine d'application. 312
7.2 Comparaison entre stratégies de prétraitement pour la binarisation de l'image de texte : (a) image originale ; (b) seuillage direct par la méthode d'Otsu ; (c) fond estimé par filtrage gaussien ; (d) image après normalisation du fond (division par l'image lissée) ; (e) CLAHE suivi d'un seuillage par Otsu ; et (f) normalisation du fond suivie d'un seuillage par Otsu. Les images illustrent l'effet de chaque technique sur la compensation des variations d'éclairage et sur la qualité de la segmentation. 313

7.3 Comparaison du texte extrait par Tesseract à partir de quatre versions de la même image : (a) image originale ; (b) CLAHE suivi d'un seuillage par Otsu ; (c) normalisation du fond suivie d'un seuillage par Otsu ; et (d) normalisation du fond en niveaux de gris, sans seuillage. La comparaison montre que la binarisation externe ne favorise pas toujours la reconnaissance, car Tesseract effectue déjà sa propre binarisation adaptative en interne. 316
7.4 Flux de reconnaissance et de traduction automatique. L'image prétraitée par normalisation du fond, sans seuillage, est utilisée comme entrée pour le Tesseract OCR, et le texte reconnu est traduit de l'anglais vers le portugais à l'aide de la bibliothèque *deep-translator*. 318
7.5 *Pipeline* d'ingestion de documents : rasterisation adaptative de pages PDF en matrices discrètes au format PNG, affichant la page deux. 320
7.6 Simulateur interactif de correction d'inclinaison (*deskew*) : déplacez le *curseur* pour incliner le document et observez les trois étapes du *pipeline* — image inclinée, contours Canny et résultat corrigé. 323
7.7 *Pipeline* de redressement axial : affichage comparatif entre l'entrée rotée d'origine, la carte de gradients structurels de Canny et le résultat final aligné avec un fond normalisé en blanc. . . 324
7.8 Détection des disques marqueurs, extraction des contours et redressement par transformation de perspective. 327
7.9 Simulateur interactif de filtrage par circularité : déplacez le *curseur* pour ajuster le seuil C et observez quels composants sont acceptés (vert) ou rejetés (rouge). 328
7.10 *Pipeline* de traitement du *QRCode* : seuillage, ouverture morphologique, inversion et recadrage final pour décodage. 331
7.11 Simulateur interactif du *pipeline* d'isolation du QRCode : parcourez les étapes de filtrage, ajustez l'élément structurant de la fermeture morphologique et voyez la détection du contour carré sur l'image originale du gabarit. 332
7.12 Décodage de code-barres linéaire avec *pyzbar* : image d'entrée et données extraites. 334
7.13 Image de la feuille de réponses en niveaux de gris chargée à partir du PDF rasterisé. 336
7.14 Zone de réponses extraite par *getAnswerArea* : région rectifiée contenant les cases de marquage.337
7.15 Région du *QRCode* isolée par *segmentQRcode* dans la zone de réponses rectifiée. 338
7.16 Recorte inférieur de la zone de réponses, concentrant les cadres de bulles à segmenter. 339
7.17 Réponses lues automatiquement par MCTest après segmentation et classification de toutes les bulles. 341
7.18 Détection de défaut par soustraction d'image : produit de référence, image avec défaut simulé et masque d'anomalie détectée. 343
7.19 Simulateur interactif d'inspection industrielle par soustraction d'images : contrôlez les distorsions géométriques de désalignement (enregistrement) et le seuil de détection pour observer l'impact sur les faux positifs. 344
7.20 Détection d'hétérogénéité de texture : carte de variance locale et masque d'anomalie. 346
7.21 Simulateur EP06_01 : IoU entre masque de référence et masque prédit 353
7.22 Simulateur EP06_02 : Filtre de Marqueurs par Circularité 355
7.23 Simulateur EP06_03 : Classification des marquages OMR 357
7.24 Simulateur EP06_04 : Estimateur de Pente par Médiane Angulaire 359
7.25 Simulateur EP06_05: Normalisation du fond par division 361
7.26 Simulateur EP06_06 : Carte de variance locale pour la détection de texture 363
7.27 Simulateur EP06_07 : *Pipeline* d'inspection — Enregistrement par translation et soustraction 365
7.28 Simulateur EP06_08 : Segmentation géométrique + Vérification par décodage de QRCode . . 369

8.1 Exemple motivationnel : différents fruits formant des regroupements distincts dans un espace de caractéristiques. 374
8.2 Aperçu du *pipeline* classique de classification d'images : extraction de descripteurs (LBP, HOG), formation de l'espace des caractéristiques et classification via k-NN. Ne s'applique pas aux modèles d'apprentissage profond (CNN, YOLO), qui apprennent des *features* de bout en bout directement à partir des pixels. **Source :** élaboré avec l'aide de Gemini Notebook ({GOOGLE}, 2025). 375

8.3 Comparaison visuelle de différents descripteurs appliqués à la même image. Chaque descripteur révèle des aspects distincts de la scène. 377
8.4 Histogramme des codes LBP de l'image du cameraman. Chaque barre représente la fréquence relative d'un code LBP, formant le vecteur de caractéristiques utilisé pour décrire la texture. 378
8.5 Premiers 100 composants du vecteur de caractéristiques HOG. 380
8.6 Importância da normalização das características para o classificador k-NN. 383
8.7 Carte conceptuelle du processus de classification d'images utilisant des descripteurs classiques (LBP, HOG, pixels bruts) et un classifieur traditionnel (k-NN). Ne s'applique pas aux modèles d'apprentissage profond (CNN, YOLO), qui apprennent des *features end-to-end* directement à partir des pixels. 384
8.8 Matrices de confusion obtenues par le classifieur k-NN utilisant trois descripteurs différents. Les lignes représentent la classe réelle (Pomme, Banane et Orange) et les colonnes la classe prédite. Plus la concentration de valeurs sur la diagonale principale est élevée, meilleure est la performance du descripteur. 387
8.9 Comparaison détaillée de l'exactitude globale dans un scénario réaliste. Observez comment les descripteurs extraits structurellement (LBP et HOG) surpassent l'utilisation des intensités pures des pixels bruts. 388
8.10 Illustration du processus de génération de données synthétiques avec *make_classification*. À gauche, visualisation des trois classes dans un espace bidimensionnel formé par deux caractéristiques informatives. À droite, importance relative de chaque caractéristique pour la classification, montrant que seulement 7 des 10 caractéristiques sont effectivement discriminantes, tandis que les autres sont redondantes (2) ou répétées (1). 391
8.11 Classification d'un nouvel échantillon par l'algorithme k-NN. Le point de test (étoile) est classé à partir des trois voisins les plus proches, mis en évidence par des cercles. 394
8.12 Simulateur interactif de la frontière de décision du k-NN : ajoutez des points d'entraînement et ajustez la valeur de k pour observer l'effet sur la région de décision. 395
8.13 Échantillon de chiffres manuscrits de la base *load_digits*, utilisé comme étude de cas de classification. 396
8.14 Matrice de confusion du classificateur k-NN entraîné avec des vecteurs d'intensité bruts (pixels).397
8.15 Comparaison de précision entre les descripteurs de pixels bruts et HOG pour le classificateur k-NN (k=3) sur la base de chiffres. 398
8.16 Précision moyenne par validation croisée (5 partitions) en fonction du paramètre k, pour la base de chiffres avec vecteurs d'intensité bruts. 400
8.17 Précision sur les ensembles d'entraînement et de test pour différentes valeurs de k. Les régions colorées représentent, de manière conceptuelle, les tendances de comportement du classifieur : risque plus élevé de surajustement (rouge), compromis entre biais et variance (vert) et risque plus élevé de sous-ajustement (bleu). 402
8.18 Échantillons synthétiques des trois classes de texture utilisées dans l'expérience de classification, générés avec un bruit gaussien et une variabilité intra-classe. 404
8.19 Précision moyenne obtenue par validation croisée (5-fold) pour les descripteurs de Pixels Bruts, LBP et HOG appliqués à la base de textures synthétiques. 406
8.20 Métriques d'évaluation détaillées pour le classificateur k-NN avec descripteurs LBP 409
8.21 Simulateur EP07_01 : Classifieur k-NN Pas à Pas 415
8.22 Simulateur EP07_02: Effet de la Normalisation *Z-score* sur la Distance k-NN 418
8.23 Simulateur EP07_03 : Précision x Rappel 420
8.24 Simulateur EP07_04 : Code LBP d'un voisinage 3×3 422
8.25 Simulateur EP07_05 : Histogramme HOG d'une cellule (mapping des angles vers les bins) . . 425
8.26 Simulateur EP07_06 : *Pipeline* k-NN Multi-Classe (vote, départage et matrice de confusion) . 427
8.27 Mosaïques de référence (format PGM ASCII) utilisées dans l'EP07_07. Cas 1 : grille 2x2 avec un échantillon de chaque classe. Cas 2 : grille 3x3 avec classes répétées et plus grande variabilité.432
8.28 Simulateur EP07_07 : Classification d'une mosaïque de textures via LBP + k-NN 433

9.1 Vue d'ensemble des principaux concepts abordés dans ce chapitre, incluant la correspondance de caractéristiques, la détection d'objets, la segmentation d'images et leurs relations avec les approches modernes basées sur le *Deep Learning*. **Source :** élaboré avec l'aide du Gemini Notebook ({GOOGLE}, 2025). . . . 436
9.2 Comparaison des performances de différentes implémentations de la distance de Hamming. . . 440
9.3 Simulateur interactif du descripteur ORB : explorez la logique de rotation et de construction du descripteur binaire. Modifiez la rotation pour observer comment le motif d'échantillonnage des tests binaires du BRIEF (lignes vertes et rouges) s'oriente dynamiquement pour garantir l'invariance angulaire. . . . 440
9.4 Image originale générée par Gemini et version pivotée (20°), simulant un changement de point de vue. . . . 441
9.5 Les 5 meilleures correspondances de caractéristiques ORB entre l'image originale et la scène synthétique, avant le filtrage par RANSAC. . . . 443
9.6 Simulateur interactif de l'algorithme RANSAC : ajustez le seuil de distance et exécutez l'algorithme pour observer la séparation entre *inliers* et *outliers*. . . . 444
9.7 Simulateur interactif du RANSAC appliqué à l'enregistrement d'images : les points clés de deux images sont appariés par un descripteur, certaines correspondances sont erronées (*outliers*), et le RANSAC estime la transformation de similarité qui aligne la plupart d'entre elles. . . . 445
9.8 Correspondances classées comme *inliers* (vert) par RANSAC lors de l'estimation de l'homographie entre les deux images. . . . 446
9.9 Enregistrement de l'image de la scène en utilisant l'homographie inverse estimée par RANSAC. 447
9.10 Détection de visages et d'yeux avec *Haar Cascade*. . . . 450
9.11 Effet de la Suppression des Non-Maximums (NMS) : les détections redondantes multiples (à gauche) sont réduites à une détection par objet (à droite). . . . 452
9.12 Segmentation classique (non basée sur l'apprentissage profond) : masque sémantique (pièce x fond) via seuillage d'Otsu, et segmentation d'instances via étiquetage des composants connexes. 455
9.13 Séparation des pièces qui se touchent via transformée de distance + *watershed*, sans érosion du contour des pièces. . . . 457
9.14 *Cadres englobants* reconstruits à partir du fichier d'annotations YOLO exporté par *mm.saveMeasures*, superposés à l'image originale des pièces. . . . 461
9.15 Simulateur EP08_01 : Distance de Hamming entre deux descripteurs binaires . . . 469
9.16 Simulateur EP08_02: RANSAC — Vote par Inliers entre Modèles Candidats . . . 471
9.17 Simulateur EP08_03 : Somme Rectangulaire en O(1) — Multiples Situations de Bord . . . 473
9.18 Simulateur EP08_04: IoU et suppression des non-maximums . . . 475
9.19 Simulateur EP08_05 : Étiquetage des composants connexes — Connectivité 4 vs. 8 . . . 477
9.20 Simulador EP08_06 : Extraction de *Bounding Boxes*, Centroides et Propriétés avec *mm.measure* 480
9.21 Simulateur EP08_07: Morphologie avec Connectivité Configurable et Mesure . . . 482
9.22 Simulateur EP08_08 : Filtrage Médian en Niveaux de Gris et Mesure d'Objets OpenCV . . . 484
9.23 Simulateur EP08_09 : Gradient d'éclairage, seuil adaptatif et mesure OpenCV . . . 485
9.24 Simulateur EP08_10: Séparation de Disques Tangents via Transformée de Distance L2 et *Watershed* . . . 487
9.25 Simulateur EP08_11 : Classification géométrique avec contrôles indépendants des *overlays BBox* (GT et DET) . . . 490
9.26 Simulateur EP08_12 : Image utilisée dans l'EP08_12. À gauche, l'image originale. À droite, l'image avec les *bounding boxes* du fichier de correction. . . . 493
9.27 Simulateur EP08_12 : Précision de Segmentation sur Objets Multiples (IoU) . . . 494

10.1 Vue d'ensemble des principaux concepts abordés dans ce chapitre. **Source :** élaboré avec l'aide du *Gemini Notebook* ({GOOGLE}, 2025). . . . 496
10.2 Simulateur interactif de convolution 2D classique : choisissez parmi les trois images synthétiques d'entrée 12×12 (maison, visage heureux ou triste) et un filtre 3×3 (Sobel V, Sobel H, Netteté ou Identité) et avancez pas à pas pour observer comment les produits internes locaux du champ réceptif construisent la carte de caractéristiques cellule par cellule. . . . 500

10.3 Simulateur interactif du neurone artificiel dans un contexte de CNN : alternez entre un neurone de couche convolutive (où x_i sont des intensités de pixel dans un champ réceptif et w_i sont des poids du *kernel*) et un neurone de couche entièrement connectée, en ajustant les entrées, les poids, le biais et la fonction d'activation pour visualiser le calcul de z et de la sortie y en temps réel. 501
10.4 Simulateur interactif de la couche convolutive : choisissez entre trois images d'entrée 12×12 (maison, visage souriant ou visage triste) pour observer comment les mêmes *noyaux* fixes réagissent à différentes contours et formes. Naviguez dans le champ réceptif avec les boutons ou la lecture automatique, ajustez la fonction d'activation et alternez le zero-padding, en suivant la carte de caractéristiques révélée cellule par cellule, avec le calcul détaillé terme à terme et la formule de la dimension de sortie en temps réel. 503
10.5 Simulateur interactif de la fonction d'activation ReLU : faites glisser le curseur pour voir comment les valeurs négatives sont mises à zéro et les valeurs positives sont préservées, à la fois sur la courbe et sur une carte de caractéristiques réelle. 504
10.6 Simulateur interactif de *pooling* : choisissez entre *max-pooling* et *average-pooling* et avancez pas à pas pour observer la réduction de la résolution spatiale de la carte de caractéristiques. . 505
10.7 Simulateur interactif de la Fonction de Perte (*Cross-Entropy*) : sélectionnez la classe réelle de l'image (Maison, Heureux ou Triste) et ajustez les probabilités estimées par Softmax pour visualiser le calcul de la pénalisation scalaire et le graphique du logarithme négatif en temps réel. 506
10.8 Simulateur interactif de *Rétropropagation* : avancez dans les étapes de la Règle de la Chaîne pour suivre le flux du signal d'erreur dans le sens inverse du réseau, en observant le calcul des dérivées partielles du gradient à chaque couche. 507
10.9 Simulador interactif des algorithmes d'optimisation : comparez la trajectoire du SGD et de l'Adam sur une surface de perte non convexe avec carte de chaleur et courbes de niveau. La ligne continue montre l'optimiseur sélectionné avançant pas à pas ; la ligne pointillée montre, pour une comparaison instantanée, le chemin complet que l'autre optimiseur parcourrait depuis le même point initial. Observez comment le SGD reste bloqué au Minimum Local à droite, tandis que l'Adam peut ou non franchir la barrière centrale selon l'impulsion accumulée et le taux d'apprentissage. Cliquez sur n'importe quel point de la carte pour réinitialiser le point initial des poids. 508
10.10 Architecture simplifiée d'une CNN pour la classification d'images, mettant en évidence le *forward pass* et les étapes d'entraînement via la fonction de perte, la rétropropagation et l'algorithme d'optimisation. 509
10.11 Simulateur interactif de l'architecture d'un CNN : choisissez l'une des images d'entrée 12×12 (maison, visage heureux ou triste), cliquez sur chaque bloc du *pipeline* — Entrée, *Conv+ReLU*, *Pooling*, *Flatten*, *FC* et *Softmax* — et exécutez de vraies étapes d'entraînement (*forward pass* + rétropropagation) pour observer la perte et la précision évoluer, les *kernels* être ajustés et le *Softmax* commencer à pointer vers la classe correcte. 510
10.12 Simulateur interactif du calcul du gradient d'un poids du *kernel* convolutif. 513
10.13 Échantillons de chiffres de l'ensemble d'entraînement après conversion en *tensors* PyTorch et normalisation. 516
10.14 Représentation du flux des transformations dimensionnelles des *tensors* à travers l'architecture *CNNDigitos*. 516
10.15 Courbes d'entraînement et d'évaluation du CNN sur la base de chiffres : évolution de la perte d'entropie croisée sur l'ensemble d'entraînement et de la précision sur l'ensemble de test au fil de 50 époques. 520
10.16 Fluxo de activations de la CNN entraînée lors du traitement d'un exemple réel du chiffre 3, du *dataset load_digits* : dimensions des tenseurs couche par couche, de l'entrée au *logit* de sortie. 521
10.17 Graphe computationnel de la *CNNDigitos* généré via *torchviz*, montrant les opérations de *forward* et les nœuds de gradient (*backward*) associés à chaque paramètre entraînable. 522
10.18 Graphe détaillé de la classe CNNDigitos généré via *torchviz*. 524
10.19 Comparaison de précision entre les classificateurs classiques du Chapitre 7 (pixels bruts et HOG avec k-NN) et le CNN entraîné dans ce chapitre, sur la même base de chiffres. 526

10.20 Échantillons des ensembles d'entraînement après prétraitement et ajustement dimensionnel aux tenseurs PyTorch : Domaine A (chiffres 0 à 4, tâche source) et Domaine B (chiffres 5 à 9 réindexés en 0 à 4, tâche cible). . . . 529
10.21 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. . . . 533
10.22 Flux d'activations et transformations dimensionnelles des tenseurs dans le modèle de transfert d'apprentissage lors du traitement d'un échantillon de test du Domaine B (chiffre 7, réindexé en classe 2). . . . 534
10.23 . . . 535
10.24 . . . 537
10.25 . . . 537
10.26 Échantillons synthétiques du *dataset* de diagnostic foliaire : feuille saine, feuille malade (taches sombres) et symptôme inconnu (chlorose jaunâtre) avec bruit sel et poivre. . . . 541
10.27 . . . 543
10.28 . . . 544
10.29 . . . 547
10.30 Échantillons initiaux du *dataset* synthétique bruité d'objets géométriques avec les boîtes englobantes du format YOLO superposées. . . . 550
10.31 Comparaison entre l'image originale avec bruit de type sel et poivre et l'image après application du Filtre Médian (3x3). Les *bounding boxes* au format YOLO restent inchangées. . . . 552
10.32 . . . 555
10.33 . . . 557
10.34 Échantillons du jeu de données synthétique de nodules : image en niveaux de gris sous bruit et masque binaire de référence correspondant. . . . 558
10.35 Ligne de base classique de segmentation : lissage, seuillage d'Otsu et ouverture morphologique affichés. . . . 560
10.36 Évolution de l'entraînement de l'U-Net compacte : réduction de la perte et élévation de l'indice IoU moyen sur l'ensemble de validation au fil de 35 époques. . . . 564
10.37 Comparativo quantitatif (IoU moyen) et qualitatif entre l'approche classique et l'U-Net entraînée sur trois échantillons de validation. . . . 565
10.38 . . . 568
10.39 . . . 569
10.40 . . . 570
10.41 Réalité augmentée basée sur un marqueur ArUco : marqueur inséré dans une scène réelle et tourné, détecté et remplacé par une image virtuelle via homographie. . . . 573
10.42 Mesure de dimensions physiques réelles à l'aide d'une carte de référence d'échelle connue (8,56 cm) rendue. Le rectangle gris représente la carte de référence et le rectangle bleu représente l'objet cible. Les dimensions estimées de l'objet sont affichées en centimètres. . . . 575
10.43 Simulateur EP09_01 : Convolution 2D manuelle (corrélation croisée + biais + ReLU, avec *padding* et *stride* ajustables) . . . 584
10.44 Simulateur EP09_02 : Pooling manuel (maximum vs. moyenne, avec fenêtre k et pas s réglables) 587
10.45 Simulateur EP09_03 : Comptage de paramètres — Convolution vs. couche entièrement connectée 592
10.46 Simulateur EP09_04 : IoU et suppression des non-maximums (NMS) . . . 595
10.47 Simulador EP09_05: Évaluation de la segmentation — IoU et Dice pixel par pixel . . . 598
10.48 Simulateur EP09_06 : Pipeline Intégré — Détection à la Mesure du Monde Réel . . . 601

A.1 Question de l'échiquier générée par MCTest, illustrant l'énoncé paramétré avec la valeur de `height` tirée pour la variante et l'exemple d'entrée/sortie du premier cas de test. . . . 608
A.2 Question réelle générée par MCTest pour le Simulado 4 — sujet « Opérateurs morphologiques ». 610

Liste des tableaux

1 Différents formats de publication générés automatiquement à partir du même code source. . . 4
2 Nombre de pages du PDF et temps de rendu par combinaison (parallélisme réel moyen de ~5 processus, pic de 8). La combinaison de base Python/Portugais ne fait que rendre les sorties déjà enregistrées dans les *notebooks* source ; les autres réexécutent tout le code. 4

1.1 Connexion entre le PDI, la VC et d'autres domaines scientifiques. 5
1.2 Principaux types d'image numérique et leurs ensembles respectifs de valeurs possibles pour chaque pixel. 9
1.3 Principales bibliothèques et modules utilisés tout au long de ce livre pour le calcul matriciel, le TNI, la VC, la visualisation et la validation des exercices. 9

2.1 Conventions d'échelle et ordre des canaux dans les principales bibliothèques de traitement d'images. 52
2.2 Comparatif entre les stratégies de compression et l'efficacité du traitement. 59
2.3 Comparatif des métriques de distance appliquées à la maille de pixels. 60
2.4 Principaux formats de stockage d'images numériques et leurs applications en TNI. 60
2.5 Méthodes d'interpolation disponibles dans mm.resize et leurs nombres respectifs de voisins utilisés dans le calcul. 66

3.1 Algorithme d'égalisation d'histogramme. 104
3.2 Interprétation typique des coefficients du *noyau*. 112
3.3 Étapes de l'*Unsharp Masking*. 126
3.4 Moyenne vs. médiane avec un pixel corrompu. La médiane ignore la valeur aberrante ; la moyenne est décalée d'environ 40 niveaux. 129

4.1 Techniques de prétraitement évaluées pour améliorer la séparation entre les pièces et le fond avant l'application de la méthode d'Otsu. 162
4.2 Propriétés de l'ouverture et de la fermeture. 173
4.3 Séquences du filtre séquentiel alterné mm.asf. 174
4.4 Comparaison entre les opérateurs *clohole* et *edgeoff*. 182
4.5 Taxinomie simplifiée des principales approches de segmentation et d'affinement étudiées dans ce chapitre. 189
4.6 *Pipeline* complet du *watershed* basé sur des marqueurs pour les images réelles. 199
4.7 *Pipeline* simplifié utilisé dans l'exemple didactique de la Figure 4.25. 202
4.8 Comparaison entre les approches basées sur les composants connexes et sur les contours. . . . 204

6.1 Correspondance entre les régions du spectre de magnitude après application du FFT Shift. . 242
6.2 Comparaison de la complexité de la convolution directe, séparable et via la Transformée de Fourier rapide (FFT). 248
6.3 Sous-bandes produites par la Transformée *Wavelet* Discrète 2D (DWT), indiquant les filtres appliqués dans chaque direction et le contenu prédominant de chaque composante. 260
6.4 Comparaison entre familles d'*ondelettes*, mettant en évidence la longueur du support, le nombre de moments nuls, la symétrie et les applications typiques. 261
6.5 Comparaison entre la transformée de Fourier discrète (TFD) et la transformée en *ondelettes* discrète (TOD), mettant en évidence leurs principales caractéristiques et applications. 270
6.6 Catégories de redondance dans les images numériques et leurs mécanismes d'exploitation respectifs. 271

6.7 Étapes du *pipeline* de compression JPEG et leurs fondements de conception respectifs. 275
6.8 Comparaison structurelle entre les principaux formats d'image matriciels. 279
6.9 Synthèse des étapes du *pipeline* de compression JPEG et de leurs impacts respectifs. 286

7.1 Images de skimage.data utilisées dans les expériences de ce chapitre. 310
7.2 Sélection d'images publiques représentatives disponibles dans le module *skimage.data*, organisées par domaine d'application. 311

8.1 Ensemble de descripteurs chromatiques, texturaux et géométriques utilisés pour représenter des images de fruits. 388
8.2 Exemple simplifié de classification de fruits utilisant deux caractéristiques normalisées. 391

9.1 Comparaison des principaux paradigmes de segmentation visuelle. 453
9.2 Architectures représentatives d'Apprentissage Profond pour la détection et la segmentation d'images. 462

10.1 Comparatif entre les principales tâches de VC selon la granularité de l'information produite. . 496
10.2 Comparaison entre les trois scénarios de transfert d'apprentissage présentés dans cette section. 545

C.1 Expressions de filtrage d'URL utilisées dans la configuration du SEB. 613

TNI & VO — Python

Bienvenue dans le manuel TNI et Vision par Ordinateur — version Python / Français.

Organisation

- **Partie I — Fondements du TNI** — Représentation, histogrammes, filtrage, morphologie
- **Partie II — Vision par Ordinateur** — Segmentation, descripteurs, détection, apprentissage profond

Préface

Ce livre constitue une **œuvre en permanente actualisation** dans les domaines du **Traitement Numérique d'Images (TNI)** et de la **Vision par Ordinateur (VO)**, conçue comme matériel didactique interactif pour les cours de premier et de deuxième cycle en Informatique, Ingénierie et domaines connexes.

> **!** Point important
>
> L'ouvrage repose sur la méthodologie décrite dans Zampirolli et al. (2025) — extension de Zampirolli et al. (2024), travail primé dans la filière **Ressources et Environnements Éducatifs** de l'**EduComp 2024**. Le contenu intègre la bibliothèque `morph.py`, développée par l'auteur.

Ce n'est pas un livre statique. Son contenu évolue continuellement au fur et à mesure que les exemples sont améliorés, que de nouvelles sections sont incorporées et que les approches pédagogiques sont affinées en fonction de l'expérience d'utilisation et des retours des étudiants et des enseignants. De cette manière, l'ouvrage est traité comme un *projet en constante évolution*, cherchant à accompagner à la fois les avancées technologiques et les meilleures pratiques d'enseignement en TNI et VO.

Contexte de la première édition

Le contenu de cette édition a été élaboré entre mai et août 2026, lors de la première offre du cours de Traitement Numérique d'Images basé sur ce matériel didactique. Le cours a été dispensé dans deux groupes de premier cycle — majoritairement composés d'étudiants en Informatique, en période matinale — et dans un groupe de deuxième cycle en Informatique, tous à l'UFABC. Cette première édition représente le résultat de cette offre initiale, consolidant le contenu développé, testé et continuellement perfectionné tout au long de la période académique.

La méthodologie d'enseignement adoptée a suivi une séquence structurée à chaque cours. Initialement, une courte vidéo était diffusée, d'une durée moyenne de 7 minutes, générée dans **Gemini Notebook**, alternant entre la présentation de la partie conceptuelle du chapitre et la résolution des Exercices de Programmation (EPs). Dans ce second cas, presque tous les EPs étaient résolus pendant le cours lui-même. Ensuite, un ensemble d'environ 12 *diapositives*, également générées dans **Gemini Notebook**, était présenté, ayant comme sources le PDF complet du livre et le fichier `morph.py`. Ces *diapositives* étaient produites à partir d'un *prompt* spécifique pour la génération du contenu théorique et pratique de chaque chapitre.

Après cette étape initiale, il restait environ 100 minutes de cours pour l'exploration des deux *notebooks* Colab du chapitre, l'un théorique et l'autre pratique, avec un accent sur les simulateurs interactifs et l'exécution des blocs de code, permettant aux étudiants de modifier les paramètres et d'observer leurs effets. Dans les cours réalisés en laboratoire, les étudiants transféraient les réponses des EPs développées dans Colab directement vers les activités VPL de Moodle, où elles étaient soumises à l'évaluation automatique. Ce processus de publication et d'utilisation des EPs est présenté à la fin de cette préface.

L'évaluation continue a inclus des simulations bimensuelles, appliquées avec le **SEB (*Safe Exam Browser*)** sur Moodle, contenant des EPs similaires à ceux des listes d'exercices. Chaque simulation accordait un bonus de 5 % sur la note finale. Les deux examens du cours utilisaient également le SEB, mais étaient composés de questions paramétrées générées dans **MCTest**, dont la description combine LaTeX et des paramètres au format `[[code:variable]]`, définis dans des extraits de Python délimités par `[[def: ... ]]` dans la question elle-même. Ce processus a permis de générer 110 variations d'examen, tirées individuellement parmi les étudiants.

L'Annexe A détaille la création de ces questions dans MCTest et leur exportation vers Moodle ; l'Annexe B décrit la configuration des activités VPL, y compris le processus de publication des EPs ; l'Annexe C présente la configuration du SEB pour les simulations et les examens ; l'Annexe D décrit la génération des vidéos et *diapositives* utilisées dans les cours avec **Gemini Notebook** ; et l'Annexe E détaille l'utilisation de l'**Intelligence Artificielle (IA)** dans la génération de *feedback* socratique pour les activités formatives et évaluatives, complétant la correction automatique réalisée par le VPL.

Comment ce livre est produit

Le contenu de ce livre est développé dans **Quarto** et stocké dans le dossier `all` du dépôt **github.com/fzampirolli/pdi-vc**. À partir d'un code source unique, le livre est automatiquement généré et publié dans différents formats, présentés dans le Table 1.

Bien que l'édition en PDF enregistre l'état de l'œuvre à la fin de la première offre du cours en 2026, le développement du livre se poursuit de manière permanente. À chaque mise à jour du dépôt GitHub, de nouvelles versions des pages HTML, du PDF et des notebooks sont automatiquement générées, mettant immédiatement les améliorations à la disposition des lecteurs.

Tableau 1: Différents formats de publication générés automatiquement à partir du même code source.

Format	Description
HTML	Version pour le *web*, avec simulateurs interactifs et navigation entre les chapitres : fzampirolli.github.io/pdi-vc/
PDF	Version adaptée à l'impression ou à la lecture *hors ligne* : livro.pt.py.pdf
Notebooks (`.ipynb`)	Compatibles avec **Jupyter** et **Google Colab**, permettant d'exécuter, de modifier et d'expérimenter les exemples de code et les Exercices de Programmation (EPs). Un filtre personnalisé maintient les références croisées, la numérotation des figures et des tableaux, et formate automatiquement les citations selon la norme ABNT.

L'ouvrage est publié en **dix combinaisons** — chaque piste de code (**Python** et **C++**) dans cinq langues (portugais, anglais, français, espagnol et italien). Avec le cache de traduction déjà rempli, une régénération complète (traduction, réexécution des *notebooks*, rendu HTML et PDF, et publication) prend environ **72 minutes**, avec un parallélisme réel moyen de ~5 processus (pic de 8) ; la **première** génération d'une nouvelle langue, avec un cache vide, est bien plus lente — environ **80 minutes** par combinaison, comme observé lors des premières générations en espagnol et en italien. Le Table 2 résume chaque combinaison (cache déjà rempli) : nombre de pages du PDF et temps de rendu. Ce temps total réel est bien inférieur à la somme des temps de chaque combinaison prise isolément (plus de **5 heures** si elles étaient exécutées une par une) : sur un processeur à plusieurs cœurs, plusieurs combinaisons s'exécutent en même temps, donc le temps total n'est pas la simple somme des temps individuels.

Tableau 2: Nombre de pages du PDF et temps de rendu par combinaison (parallélisme réel moyen de ~5 processus, pic de 8). La combinaison de base Python/Portugais ne fait que rendre les sorties déjà enregistrées dans les *notebooks* source ; les autres réexécutent tout le code.

Combinaison	Piste	Langue	Pages	Temps de rendu
`py.pt`	Python	Portugais (base)	654	~7 min
`py.en`	Python	Anglais	644	~31 min
`py.fr`	Python	Français	666	~31 min
`py.es`	Python	Espagnol	666	~31 min
`py.it`	Python	Italien	658	~31 min
`cpp.pt`	C++	Portugais	434	~26 min
`cpp.en`	C++	Anglais	426	~34 min

Combinaison	Piste	Langue	Pages	Temps de rendu
`cpp.fr`	C++	Français	434	~38 min
`cpp.es`	C++	Espagnol	430	~37 min
`cpp.it`	C++	Italien	428	~35 min

État de validation. Seule la combinaison `py.pt` (Python, portugais) est la source curée : c'est là que l'auteur écrit, révise et valide tout le contenu. Les neuf autres combinaisons — les pistes **C++** et les traductions en anglais, français, espagnol et italien — sont **générées automatiquement**, par traduction via un modèle de langage et transpilation du code, et **nécessitent encore une révision détaillée**. Le point le plus sensible est le **texte incrusté dans certaines figures** : là où la version traduite n'a pas encore été produite, l'image apparaît avec du texte en portugais (chaque figure traduite suit le même suffixe de langue que les autres fichiers générés, par exemple `image_en.png`).

Les pages HTML, le PDF et les *notebooks* disponibles dans le projet reflètent toujours l'état le plus récent du développement. Lorsqu'une **édition officielle** est publiée, elle est identifiée par une *balise* dans le dépôt GitHub, permettant de reproduire exactement le contenu correspondant à cette édition. Ainsi, le lecteur peut suivre à la fois l'évolution continue du projet et récupérer toute édition officielle précédemment publiée.

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

Chaque chapitre met à disposition deux boutons **Exécuter Colab** (Figure 1), qui dirigent vers des environnements complémentaires :

1. **Partie Théorique**, située à l'ouverture de chaque chapitre, contenant les concepts présentés et des exemples de code exécutables ;
2. **Partie Pratique**, dans la section **Partie Pratique avec Exercices de Programmation** (EPs), dédiée aux exercices et à leurs simulateurs interactifs.

Figure 1: Bouton cliquable **Exécuter Colab**, disponible au début des parties **Théorique** et **Pratique** de chaque chapitre, permettant l'exécution interactive des exemples et des exercices. Dans la version PDF, il existe un *lien* HTML direct entre l'image du simulateur et sa légende.

La génération des *notebooks* est entièrement automatisée par le *script* `gerar_notebooks_alunos.py`, qui adapte le contenu aux environnements interactifs, permettant son exécution sans nécessité d'installation de Quarto.

Utilisation d'outils d'Intelligence Artificielle

La conception, le projet pédagogique, la structure conceptuelle et le contenu fondamental de ce livre sont de **paternité exclusive de l'auteur**.

Dans le processus d'édition et de soutien au développement, des outils d'**Intelligence Artificielle (IA)**, tels que **ChatGPT**, **Claude**, **DeepSeek** et **Gemini**, ont été utilisés dans leurs versions gratuites, employés strictement comme ressources auxiliaires. Leur utilisation s'est limitée au soutien à la révision et à l'amélioration du style textuel, à l'optimisation de la syntaxe des codes et à la génération assistée d'illustrations conceptuelles et d'exemples.

Toutes les réponses et contenus suggérés par ces outils ont été soumis à une **analyse critique, une vérification, une validation technique, une adaptation et une intégration par l'auteur**, qui assume la responsabilité du texte, des codes et des ressources pédagogiques présentés. Les outils d'IA **ne sont pas**

considérés comme auteurs ou coauteurs de l'œuvre, ni responsables des décisions intellectuelles, techniques ou pédagogiques qui fondent son contenu.

Il convient également de souligner que les **simulateurs** et ressources interactives présents dans le livre — disponibles dans les versions HTML et Jupyter Notebook (IPYNB) — ont été conçus et implémentés dans le cadre de l'approche pédagogique de l'œuvre, dans le but de permettre une expérimentation directe des concepts présentés et de favoriser l'autonomie d'apprentissage du lecteur.

Distinct de l'utilisation éditoriale décrite ci-dessus, le *pipeline* de génération multilingue (voir « Comment ce livre est produit ») emploie un modèle de langage comme **composant d'ingénierie du système** : la traduction automatique du contenu entre le portugais et d'autres langues, avec un cache incrémental qui évite de retraduire les passages inchangés. Cette étape utilise l'API payante de **DeepSeek** (modèle `deepseek-v4-flash`) ; le livre complet a déjà été traduit du portugais vers l'**anglais**, le **français**, l'**espagnol** et l'**italien**, et les **chapitres 1 à 5** disposent en outre d'une piste **C++** qui compile et s'exécute réellement — le tout pour un coût cumulé de l'ordre de quelques dollars, grâce au cache incrémental.

Certaines images du livre contiennent encore du texte en portugais ; elles seront également traduites à l'avenir dans les autres langues, en suivant le même modèle de suffixe que les autres fichiers générés (par exemple, `image_en.png`).

La bibliothèque `morph.py`

La bibliothèque **`morph.py`** (Zampirolli et al., 2025) accompagne toute l'œuvre comme un outil de soutien à l'enseignement. Son objectif n'est pas de remplacer les bibliothèques consolidées, comme **OpenCV**, **scikit-image**, **scikit-learn**, **NumPy** ou **Matplotlib**, ni de rivaliser avec elles en performance ou en couverture. Au lieu de cela, elle cherche à **rendre transparents les algorithmes fondamentaux du Traitement Numérique d'Images et de la Vision par Ordinateur (TNI-VO)**, permettant à l'étudiant de comprendre leur implémentation, de modifier le code et de développer de nouvelles fonctionnalités.

Chaque fois que possible, la bibliothèque offre deux versions pour un même algorithme : une **implémentation didactique**, écrite pas à pas pour refléter directement les définitions mathématiques présentées dans le livre, et une autre **implémentation optimisée**, basée sur des bibliothèques spécialisées pour les applications pratiques.

> **ℹ Héritage Technique**
>
> La structure de `morph.py` a pour base des outils de Morphologie Mathématique développés au Brésil, comme **MMachLib** (Lotufo et al., 1997) et **MMach** (Barrera et al., 1998), ainsi que la **mmorph**, utilisée dans l'ouvrage de Dougherty; Lotufo (2003). Ces bibliothèques ont servi de référence pour l'enseignement du domaine pendant des décennies.

Cette philosophie peut être observée, par exemple, dans les opérateurs morphologiques de dilatation :

- `mm.dil` : implémentation optimisée, indiquée pour les applications pratiques ;
- `mm.dil0` : implémentation didactique de la dilatation pour les **éléments structurants planaires**, suivant directement la définition classique de la Morphologie Mathématique ;
- `mm.dil1` : implémentation didactique de la dilatation pour les **fonctions structurantes (*kernels* non planaires)**, suivant rigoureusement sa formulation mathématique.

La bibliothèque offre également des fonctions pour la lecture, l'affichage, la conversion des couleurs, le filtrage et la morphologie mathématique via une interface simple :

```
from morph import mm

img = mm.gray(mm.read("lena.jpg"))  # lecture et conversion en niveaux de gris
grad = mm.gradm(img)                # gradient morphologique
mm.show(grad)                       # affichage
```

Au long du livre, de nouvelles méthodes sont introduites dans la bibliothèque pour simplifier la réalisation des Exercices de Programmation (EPs). Parmi elles figurent `mm.readTrain()` et `mm.readTest()`, qui automatisent la lecture des bases d'entraînement et de test utilisées dans les chapitres dédiés à la reconnaissance de formes.

De plus, tous les projets du **Gemini Notebook** incluent le fichier `morph.py`, permettant de consulter et de discuter directement l'implémentation des algorithmes présentés dans le livre. De cette manière, l'étudiant peut utiliser le Gemini Notebook lui-même comme assistant d'étude, en posant des questions telles que :

> Expliquez comment l'algorithme `mm.knn0` du `morph.py` a été implémenté, en montrant le code et en commentant chaque étape de manière didactique. De plus, expliquez les différences entre `mm.knn0()` et `mm.knn()`.

! Implémentations didactiques et implémentations optimisées

Dans plusieurs chapitres, un même algorithme est présenté en deux versions. Les fonctions avec suffixe `0`, `1`, etc. (par exemple, `mm.knn0()`, `mm.dil0()` et `mm.dil1()`) sont des implémentations didactiques, développées pour faciliter la compréhension des algorithmes. Quant aux fonctions sans suffixe (comme `mm.knn()` et `mm.dil()`), elles utilisent des implémentations optimisées, souvent basées sur des bibliothèques spécialisées, telles que OpenCV, scikit-image et scikit-learn, étant plus adaptées aux applications pratiques.
Dans les activités évaluées par le **VPL de Moodle**, des bibliothèques comme **scikit-learn** et, dans certains cas, **scikit-image**, peuvent ne pas être utilisées en raison des limitations de mémoire de l'environnement d'exécution. Pour cette raison, les Exercices de Programmation (EPs) emploient préférentiellement les implémentations didactiques de `morph.py`, qui produisent les mêmes résultats, bien qu'avec une performance computationnelle moindre. Cette restriction n'existe pas lorsque les programmes sont exécutés localement, par exemple dans **VS Code**, **Jupyter Notebook** ou **Google Colab**.

Le code source de la bibliothèque est disponible à :

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

Exercices de Programmation (EPs) et Validation Automatique

Chaque unité du livre inclut des **Exercices de Programmation (EPs)** pratiques et de complexité croissante, conçus pour consolider les concepts présentés tout au long du chapitre. Chaque EP est accompagné d'un **simulateur interactif**, disponible dans les versions HTML et IPYNB, qui permet à l'étudiant de manipuler les paramètres, de visualiser le comportement des algorithmes et de développer une compréhension intuitive du problème avant de commencer son implémentation. De cette manière, le processus d'apprentissage combine expérimentation, programmation et validation automatique.

La validation des solutions est effectuée localement par la classe `TestSuite` (`testsuite.py`), qui compare la sortie du programme avec les fichiers de cas de test (`.cases`) :

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

Le système prend en charge plusieurs langages (Python, Java, C++, C, JavaScript et R) et peut être intégré directement à Moodle. Pour les enseignants intéressés par le processus complet pas à pas de publication des EPs comme activités VPL, consultez le **Guide de l'Enseignant**, à la fin de cette préface, et l'Annexe B.

Code ouvert

Le projet est régi par des principes de code ouvert. Le dépôt public rassemble le texte, les codes, les images et les *scripts* : **github.com/fzampirolli/pdi-vc**

Chaque version du livre est archivée en permanence sur **Zenodo** avec un *DOI citable*. Pour citer ce matériel dans des travaux académiques :

> ZAMPIROLLI, Francisco de Assis. **PDI+VC — Traitement Numérique d'Images et Vision par Ordinateur**. UFABC, 2026. DOI : 10.5281/zenodo.20784605

Il convient de noter, à propos, que le contenu exposé dans cet ouvrage reflète le regard critique, la proposition pédagogique et l'expérience d'enseignement de son auteur. Ainsi, les analyses, approches et opinions exprimées tout au long de ce livre représentent uniquement la compréhension de son concepteur, ne constituant ni ne reflétant une position officielle ou institutionnelle de l'Université Fédérale de l'ABC (UFABC).

Avant de commencer : *Notebooks* en Python

Le concept de ***Literate Programming*** (Programmation Littéraire), proposé par Donald Knuth (Knuth, 1984), fonde la structure de ce matériel. La logique inverse le paradigme traditionnel : le programme est écrit pour la lecture humaine, ressemblant à un essai, tandis que le code est extrait séparément pour l'exécution computationnelle.

Le contenu est structuré en *notebooks* — des documents qui entrelacent des **cellules de texte** (en Markdown) et des **cellules de code** (en Python).

- **Exécution :** les cellules de code sont identifiées par `[ ]`. L'exécution peut être réalisée avec `Shift + Enter` ou par le bouton ▷ de l'interface.
- **Environnements :** les *notebooks* peuvent être exécutés localement, via Jupyter ou Visual Studio Code (VS Code), ainsi que dans des environnements cloud, comme Google Colab.

> **Note sur le format**
>
> Dans les environnements interactifs, le code peut être modifié et exécuté. Dans les versions statiques (HTML ou PDF), les blocs de code ont une finalité de lecture et de référence, sans préjudice à l'intégrité des explications.

Guide de l'Enseignant : Publication des EPs sur Moodle

*Cette section est destinée aux enseignants souhaitant intégrer les Exercices de Programmation (EPs) aux activités **VPL (Virtual Programming Lab)** de Moodle, en complément de l'Annexe B.*

Intégration avec Moodle (VPL)

Les EPs des *notebooks* peuvent être convertis en activités **VPL** sur Moodle. Le *script* `ep_tools.py` (exécuté via `make build`) automatise l'exportation des EPs de la fin de chaque chapitre en deux étapes :

1. **Extraction** (`make eps`) : génère un HTML interactif indépendant par EP, avec énoncé, exemples et simulateur, dans `gen/book/eps/<versao>/`. Voir la liste complète à : https://fzampirolli.github.io/pdi-vc/eps/py.pt/index.html
2. **Conversion** (`make moodle`) : transforme le HTML en fragment autonome, sans dépendance à un CSS externe, prêt à être collé dans l'éditeur de Moodle, dans `gen/book/eps/<versao>_moodle/`. Voir un exemple à : https://fzampirolli.github.io/pdi-vc/eps/py.pt_moodle/EP01_01.html

De plus, tous les simulateurs interactifs développés tout au long du livre peuvent être consultés directement à https://fzampirolli.github.io/pdi-vc/simuladores/py.pt/.

Autres pistes et langues

Les liens ci-dessus utilisent `py.pt` comme exemple. Pour accéder aux EPs ou simulateurs d'une autre combinaison langage/langue, il suffit de remplacer ce segment dans l'URL — par exemple, `cpp.it` pour la piste C++ en italien : `https://fzampirolli.github.io/pdi-vc/eps/cpp.it/index.html`. La structure est la même pour toutes les combinaisons disponibles, sous `/eps/<version>/`, `/eps/<version>_moodle/` et `/simuladores/<version>/`.

Lors de la publication avec `make publish`, ces **deux versions** de chaque EP sont disponibles aux *liens* indiqués. L'enseignant peut combiner les deux stratégies : coller la version Moodle dans l'éditeur VPL (avec simulateur fonctionnel) et inclure dans l'énoncé un *lien* vers la version complète, où les formules sont rendues correctement par MathJax.

Révision obligatoire avant de publier sur Moodle

Bien qu'automatisée, la conversion exige une révision de l'enseignant en raison des limitations de l'éditeur TinyMCE/HTML Purifier de Moodle :

- **Formules mathématiques** — TinyMCE rejette les barres obliques inverses (\), corrompant les équations LaTeX. Pour cette raison, la version Moodle convertit les formules simples en HTML pur (entités comme `≥`, `×`). Les formules complexes (`\begin{cases}`, matrices, intégrales) peuvent sortir incomplètes — vérifiez et, si nécessaire, réécrivez-les en texte ou indiquez la version complète via *lien*.
- **Figures externes** — Les images complémentaires doivent être insérées manuellement dans l'activité VPL.
- **Simulateurs** — Fonctionnent parfaitement à condition d'utiliser du JavaScript standard avec des styles *inline* et une manipulation directe du DOM (`getElementById`). **Attention :** si vous incluez des formules en LaTeX contenant le caractère \ sur Moodle, les simulateurs peuvent cesser de fonctionner.

Comment publier sur Moodle

1. Accédez à la version Moodle de l'EP souhaité :

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

2. Copiez le code source complet de la page (`Ctrl+U` → `Ctrl+A` → `Ctrl+C`).
3. Sur Moodle, créez l'activité VPL, accédez à l'éditeur HTML, collez le contenu (`Ctrl+V`) et enregistrez.
4. Importez les fichiers `.cases` du dossier `all/capXX/casos/` dans les paramètres de test du VPL.

Réutilisation des Cas de Test

Les fichiers `.cases` sont compatibles avec le VPL, permettant d'utiliser le même ensemble de tests tant dans le développement local que dans la correction automatisée sur Moodle.

Partie I

Partie I — Fondements du TNI

Chapitre 1

Fondements et Premiers Pas

Exécuter Colab Ouvrir GitHub

Ce chapitre inaugure la **Partie 1** de l'ouvrage, consacrée aux fondements du Traitement Numérique des Images (TNI). Y sont présentés la représentation mathématique des images numériques et les principales méthodes pour leur manipulation.

On utilise le langage Python et la bibliothèque `morph.py` (ZAMPIROLLI, 2025).

1.1 Objectifs

À la fin de ce chapitre, vous serez capable de :

- Comprendre la nature physique et mathématique de l'image numérique $f(x, y)$.
- Identifier les bandes du spectre électromagnétique pertinentes pour le TNI.
- Effectuer les opérations de base : lecture, affichage et sauvegarde d'images.
- Accéder aux intensités des pixels et les modifier individuellement.
- Appliquer un seuillage manuel.
- Configurer l'environnement de développement en Python.
- Manipuler les structures de matrices (NumPy) sans tomber dans les pièges de la mémoire.

1.2 Avant de commencer : *Notebooks* interactifs

Ce matériel a été conçu selon le concept de ***Literate Programming*** (Programmation lettrée), imaginé par Donald Knuth dans les années 1980 (KNUTH, 1984). Knuth — également créateur du système **TeX** pour la typographie numérique — a proposé que les programmes soient écrits comme un récit logique destiné aux êtres humains, entrelaçant code et documentation.

Pour exécuter une cellule, appuyez sur Shift + Enter ou cliquez sur le bouton ▷.

> **Note sur le format**
>
> Dans les versions rendues (PDF ou HTML), le code est présenté dans des blocs statiques à des fins de lecture et de référence. L'exécution interactive nécessite un accès via Google Colab (disponible en haut de la page) ou dans un environnement local via VSCode ou Jupyter Notebook.

1.3 Fondements

L'étude des systèmes basés sur l'image englobe un écosystème de disciplines intégrées qui transforment des données visuelles brutes en connaissances structurées. Tandis que certains domaines se concentrent sur la

génération de représentations, d'autres se consacrent au traitement et à l'analyse de ces données pour soutenir des applications technologiques complexes.

Le diagramme présenté dans le Figure 1.1 établit la distinction et la complémentarité entre le Traitement Numérique des Images (TNI) et la Vision par Ordinateur (VO). Le TNI, mis en évidence en **vert**, se concentre sur la transformation d'image en image, visant l'amélioration de la qualité ou le prétraitement, comme la suppression du bruit et le rehaussement du contraste.

En revanche, la VO, signalée en **bleu**, se focalise sur l'interprétation du contenu visuel pour extraire des modèles ou des informations, tels que la reconnaissance d'objets et de gestes. La région d'intersection illustre la synergie entre les domaines, où le TNI prépare les données visuelles pour l'interprétation par la VO. La carte démontre également les interconnexions des deux disciplines avec des domaines tels que la Robotique, l'Infographie, l'Intelligence Artificielle (IA) et les Neurosciences.

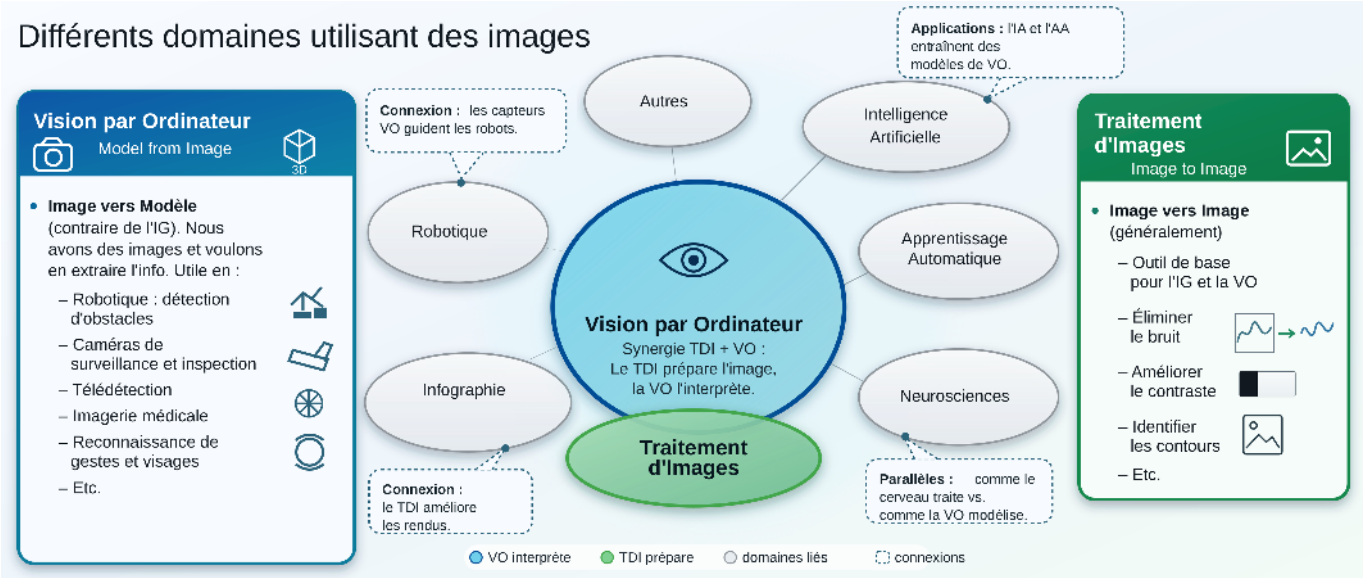


Figure 1.1: Diagramme relationnel détaillant les distinctions fondamentales, les synergies et les interconnexions entre le TNI et la VO dans le contexte des systèmes basés sur l'image.

1.3.1 👁️ Vision par ordinateur

- **Focus :** *Image* → *Modèle* (chemin inverse de l'infographie).

- **Objectif :** Extraire des informations de haut niveau à partir d'images ou de vidéos.

- **Applications typiques :**
  - Robotique – détection d'obstacles, localisation et navigation autonome.
  - Surveillance et inspection – reconnaissance d'événements, lecture de plaques, contrôle qualité.
  - Télédétection – analyse d'images satellite, cartographie environnementale.
  - Imagerie médicale – détection de tumeurs, segmentation d'organes, aide au diagnostic.
  - Interaction homme-machine – reconnaissance de gestes, expressions faciales, suivi oculaire.
- **Relation avec d'autres domaines :** utilise des techniques d'apprentissage automatique et d'IA pour classer et interpréter les scènes ; sert d'« yeux » à la robotique.

1.3.2 🖼️ Traitement Numérique des Images (TNI)

- **Focus :** *Image* → *Image* (généralement — transformation d'une image en une autre).

- **Objectif :** Améliorer la qualité visuelle ou extraire des caractéristiques de bas niveau.

- **Applications courantes :**
  - Élimination du bruit (filtres moyenne, médiane, gaussien).
  - Amélioration du contraste (égalisation d'histogramme, ajustement gamma).
  - Détection des contours (Sobel, Canny, Laplacien).
  - Segmentation (seuillage, croissance de régions, *watershed*).
  - Transformations géométriques (redimensionnement, rotation, correction de perspective).
- **Relation avec d'autres domaines (voir Table 1.1) :**
  - C'est la **base** de la plupart des systèmes de VC (prétraitement).
  - L'infographie applique souvent le TNI pour le post-traitement (ex. : lissage, rehaussement).
  - Les techniques d'IA peuvent optimiser les paramètres de traitement (ex. : apprentissage de filtres).

Tableau 1.1: Connexion entre le PDI, la VC et d'autres domaines scientifiques.

Domaine	Relation avec le PDI et la VC
Intelligence artificielle	Fournit des modèles (réseaux de neurones, SVM) qui interprètent les sorties de la VC.
Robotique	Consomme des données de la VC pour prendre des décisions (navigation, manipulation).
Apprentissage automatique	Utilise des descripteurs extraits par le PDI/VC pour entraîner des classifieurs.
Infographie	Chemin inverse : *modèle* $\rightarrow$ *image* ; applique souvent le PDI pour un rendu réaliste.
Neuroscience	Inspire des modèles de PDI (ex. : filtres similaires aux cellules ganglionnaires de la rétine).

1.4 Étapes du PDI

Les étapes du PDI sont présentées dans la Figure 1.2 et peuvent être comprises comme une chaîne de transformations qui réduit la redondance des données afin d'en extraire du sens :

- **Niveau bas :** agit directement sur les pixels de l'image bruitée pour effectuer des améliorations et des filtrages, en produisant en sortie une image propre ou rehaussée.
- **Niveau moyen :** reçoit l'image traitée, effectue la segmentation et la description, transformant la matrice de pixels en attributs structurés (forme, taille et texture).
- **Niveau haut :** utilise la table d'attributs pour alimenter des processus logiques et d'intelligence artificielle, aboutissant à la décision ou à la reconnaissance finale (comme le diagnostic médical).

A Figure 1.3 détaille la séquence complète du PDI, depuis la capture jusqu'à l'interprétation. Le flux commence à l'acquisition de l'image (1) et se poursuit avec l'amélioration (2) et la restauration (3). Ensuite, le contenu est isolé par la segmentation (4) et affiné par la morphologie (5). La transition cruciale se produit à la représentation et à la description (6), où les objets visuels sont convertis en données mathématiques (aire, périmètre, etc.), permettant la reconnaissance (7). Les processus auxiliaires incluent le traitement d'image couleur et la compression, qui contribuent à l'efficacité du stockage et de l'analyse.

1.5 Formation de l'image et le spectre

Le processus de formation d'une image repose sur l'interaction entre la matière et l'énergie rayonnante. Essentiellement, une image est conçue lorsqu'un **capteur enregistre le rayonnement** issu de l'interaction avec un objet physique. Dans le contexte de la vision humaine et de la photographie conventionnelle, ce phénomène dépend d'une source de lumière qui éclaire la scène ; les caractéristiques des objets sont alors codées par le biais des **variations d'intensité et de couleur** de la lumière qui atteint le capteur, comme illustré dans la Figure 1.4.

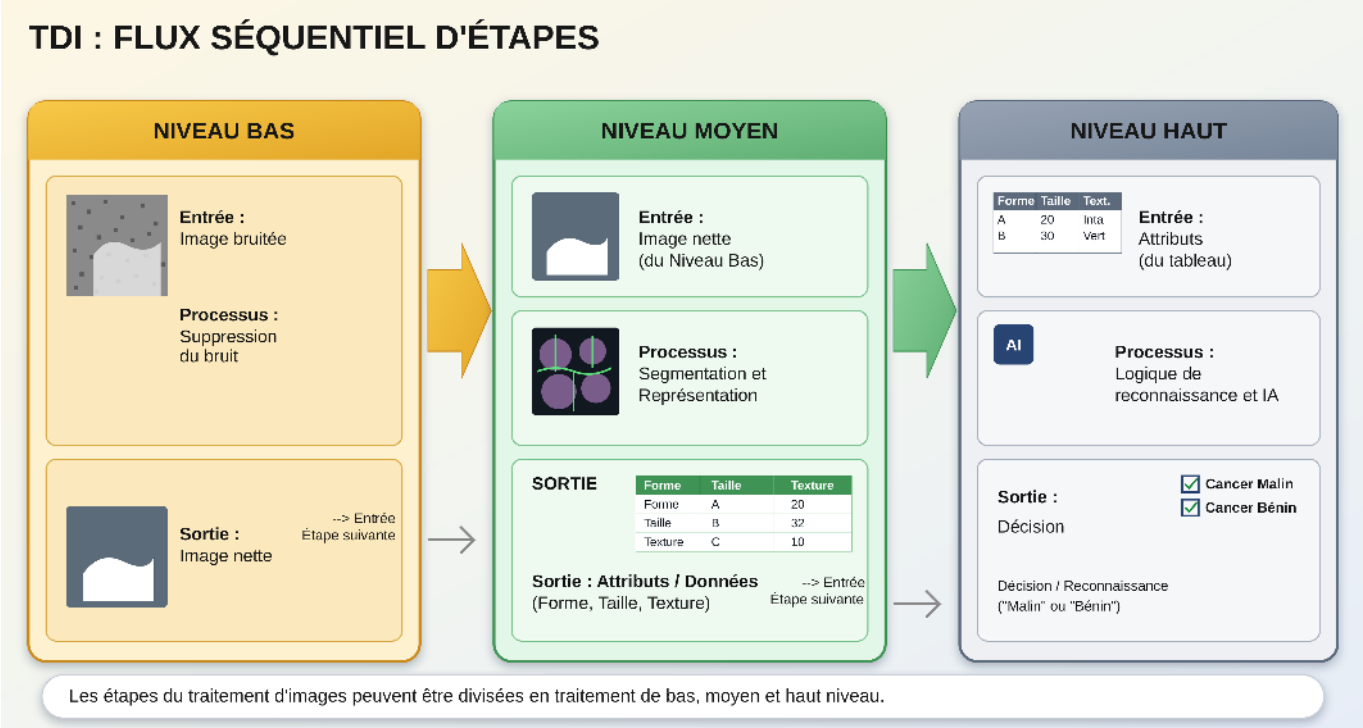


Figure 1.2: Représentation du flux séquentiel de traitement : la sortie de chaque niveau devient l'entrée du niveau suivant.

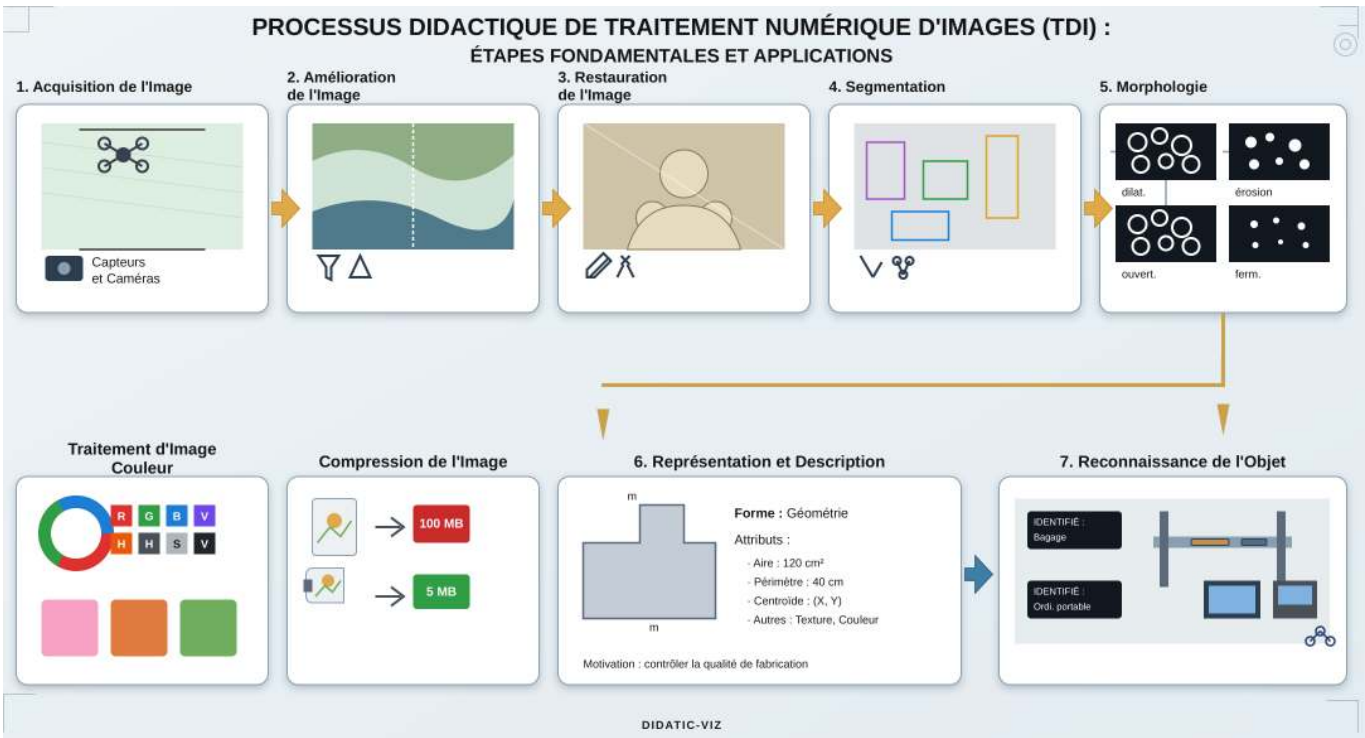


Figure 1.3: Flux détaillé du PDI : de l'acquisition sensorielle à l'extraction d'attributs et à la reconnaissance automatisée, incluant le traitement couleur et la compression.

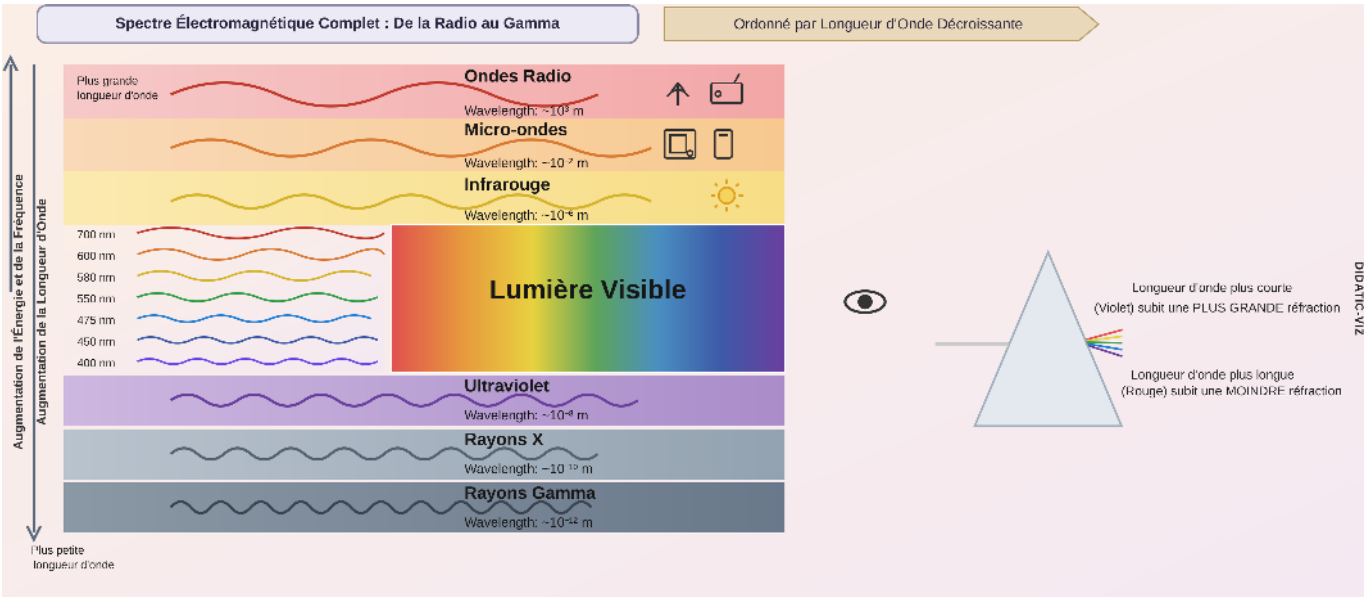


Figure 1.4: Représentation du spectre visible et de sa position par rapport aux autres rayonnements électromagnétiques, mettant en évidence la variation des longueurs d'onde de 380 nm à 750 nm.

La **lumière visible** n'occupe qu'une petite bande du spectre électromagnétique — entre **380 nm (violet)** et **750 nm (rouge)** — comme illustré dans la Figure 1.5. Les capteurs numériques classiques opèrent dans cette même fenêtre, mais des équipements spécialisés peuvent capter des rayonnements invisibles à l'œil humain, tels que l'infrarouge et les rayons X. En PDI, l'image formée dépend directement de la sensibilité spectrale du capteur utilisé.

À partir de ce processus physique d'acquisition, il devient possible de modéliser mathématiquement l'image numérique comme une fonction bidimensionnelle discrète, dans laquelle chaque point de la scène est représenté par des échantillons numériques d'intensité lumineuse, formalisant ainsi les concepts de pixel et d'image numérique présentés dans la section suivante.

1.6 Qu'est-ce qu'une image numérique ?

Une **image numérique** est constituée d'une grille de **pixels** (*Picture Elements*), où chaque pixel est la plus petite unité élémentaire de l'image.

> Qu'est-ce qu'un pixel ?
>
> Un **pixel** est la plus petite unité adressable qui compose une image numérique. Chaque pixel occupe une position unique dans la grille et stocke une ou plusieurs valeurs numériques qui représentent son intensité ou sa couleur.

Représentation mathématique

Contrairement à une fonction continue, le domaine d'une image numérique est un plan rectangulaire fini $\mathbb{E} \subset \mathbb{Z}^2$, qui représente la grille d'échantillonnage. Ce domaine est indexé par des coordonnées entières :

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

Où :

- L : représente la largeur de l'image (nombre de colonnes).
- H : représente la hauteur de l'image (nombre de lignes).

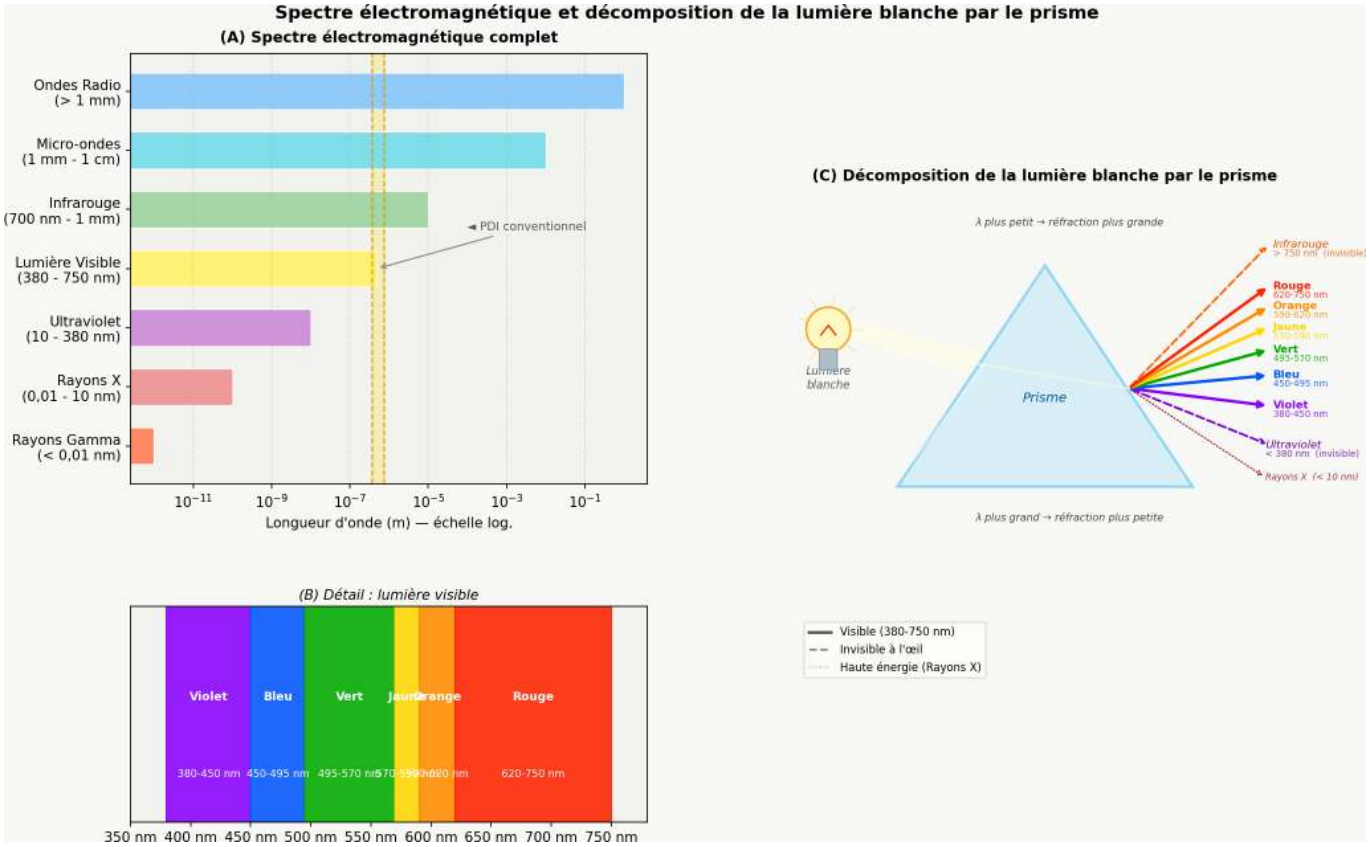


Figure 1.5: (A) Spectre électromagnétique complet en échelle logarithmique, avec mise en évidence de la bande visible. (B) Détail de la lumière visible (380-750 nm) et de ses couleurs. (C) Décomposition de la lumière blanche par le prisme : la longueur d'onde la plus courte subit une réfraction plus grande, séparant UV, visible et infrarouge.

L'image numérique est une fonction qui associe à chaque paire de coordonnées (x, y) une ou plusieurs valeurs décrivant l'apparence du pixel.

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

L'ensemble $\mathcal{V}$ définit les valeurs possibles pour le pixel (codomaine), variant selon le type d'image, comme illustré dans la Table 1.2.

Tableau 1.2: Principaux types d'image numérique et leurs ensembles respectifs de valeurs possibles pour chaque pixel.

Type d'image	$\mathcal{V}$ (valeurs du pixel)	Représentation
Binaire	$\{0, 1\}$ ou $\{0, 255\}$	■□
Niveaux de gris	$\{0, 1, \dots, 255\}$	[ ][=][#]■
Couleur (RVB)	$\{0, \dots, 255\}^3$ (triplets ordonnés de valeurs)	■■■

Exemple pratique : Une image couleur dans le modèle RVB peut être représentée mathématiquement par une fonction qui associe trois valeurs d'intensité à chaque pixel. Informatiquement, cette représentation correspond à trois matrices superposées — les canaux rouge (*Red*), vert (*Green*) et bleu (*Blue*) — dans lesquelles chaque élément stocke l'intensité lumineuse du canal respectif à une position donnée de l'image.

Pour rendre possibles les expériences pratiques de PDI et de VC, ce livre utilise l'écosystème scientifique de Python, en intégrant des bibliothèques dédiées à la manipulation matricielle, au traitement d'images et à la visualisation des résultats, présentées ci-dessous.

1.7 Configuration de l'environnement

Ce matériel utilise l'écosystème scientifique de Python, avec un accent particulier sur **NumPy** (calcul matriciel), **OpenCV** (vision par ordinateur) et la bibliothèque **morph.py** (ZAMPIROLLI, 2025), développée à des fins pédagogiques et utilisée tout au long de ce livre, comme présenté dans la Table 1.3.

Le projet est actuellement disponible en **Python** et en **portugais**. Son organisation permet toutefois son extension à d'autres langages de programmation et à d'autres langues.

Les **Exercices de Programmation (EP)** présentés à la fin des chapitres peuvent être validés par le module `testsuite.py`, qui permet d'exécuter des cas de test dans différents langages, notamment Python, C, C++, Java, JavaScript et R. Ainsi, les mêmes cas de test peuvent être utilisés pour vérifier les solutions développées dans les *notebooks* et dans **Moodle/VPL**.

Tableau 1.3: Principales bibliothèques et modules utilisés tout au long de ce livre pour le calcul matriciel, le TNI, la VC, la visualisation et la validation des exercices.

Bibliothèque	Fonction principale
`numpy`	Représentation matricielle et opérations numériques
`opencv-python`	Lecture, écriture et opérations de vision par ordinateur
`matplotlib`	Visualisation d'images et de graphiques
`morph.py`	Abstraction didactique des opérations de TNI
`testsuite.py`	Exécution et validation automatique des EP

1.8 Versions de `morph.py`

La bibliothèque `morph.py` possède deux versions publiques :

- **Version 1.0** — version originale, publiée avec l'article présenté à EduComp 2024 : dépôt de la version 1.0
- **Version 1.1** — version utilisée dans les activités de ce livre : dépôt de la version 1.1

La version 1.1 a été adaptée pour répondre également aux contraintes de mémoire du **Moodle/VPL** (Laboratoire Virtuel de Programmation). Dans la version 1.0, certaines bibliothèques, comme `matplotlib`, `requests` et `skimage`, étaient chargées au moment du `import morph`. Dans des environnements aux ressources limitées, ce chargement pouvait dépasser la limite de mémoire et produire des erreurs comme `Jail: out of memory, 128MiB`.

Dans la version 1.1, ces *imports* ont été remplacés par un **chargement *lazy*** : chaque bibliothèque n'est importée que par la méthode qui l'utilise effectivement. Ainsi, la commande `import morph` maintient un chargement initial réduit, utilisant principalement `numpy` et `cv2`, tandis que les ressources supplémentaires sont chargées à la demande.

La configuration des *notebooks* est centralisée dans le fichier `config.py`, disponible dans le même dépôt. Ce fichier vérifie les dépendances, cherche à garantir l'utilisation d'**OpenCV 5.0** et met à disposition `morph.py` et, si nécessaire, `testsuite.py`. Lorsque l'environnement ne dispose pas de la version adéquate d'OpenCV, `config.py` peut réinstaller la version requise et, si nécessaire, demander le redémarrage de l'environnement d'exécution.

Ainsi, les *notebooks* utilisent une routine de configuration unique, réduisant la répétition de code et facilitant la reproduction de l'environnement informatique utilisé dans ce livre.

```
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
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0 | TestSuite : 1.1.2
```

1.9 Fondements des matrices — attention à la copie de références

Comme une image numérique peut être représentée par une matrice, il est important de comprendre comment créer et manipuler correctement les matrices. En Python, il existe un piège courant lors de l'utilisation de l'opérateur `*` avec des listes :

> ⚠ Attention à la copie de références
>
> Lors de l'exécution de `m = [[0]*2]*3`, trois lignes indépendantes ne sont pas créées. À la place, trois références vers la **même liste** sont créées. Par conséquent, la modification d'un élément dans l'une des lignes modifiera également l'élément correspondant dans les autres lignes.

Pour visualiser ce comportement, on peut exécuter le code sur Python Tutor et le comparer avec la méthode correcte de création d'une matrice à l'aide de listes : `m = [[0]*2 for _ in range(3)]`.

En pratique, pour le traitement numérique des images (TNI), il est recommandé d'utiliser **NumPy**, qui fournit des structures de données multidimensionnelles efficaces et des opérations vectorisées adaptées à la manipulation d'images. Le code suivant présente différentes méthodes de création d'images synthétiques, dont les résultats sont affichés dans la Figure 1.6.

```python
import numpy as np

# Création d'une image noire (zéros) de 4, 6 pixels
img_preta = np.zeros((4, 6), dtype='uint8')
img_preta[0,0] = 255  # pixel blanc en haut à gauche

# Création d'une image blanche (255) de 4, 6 pixels
img_branca = np.ones((4, 6), dtype='uint8') * 255
img_branca[3,5] = 0  # pixel noir en bas à droite

# Création d'une image aléatoire pour les tests (bruit)
img_random = mm.randomImage(4, 6, maxValue=255)

print("Matrice aléatoire générée :")
print(mm.drawImg(img_random))

mm.show(
    [img_preta, img_branca, img_random],
    titles=[
        "Principalement noire\n(avec 1 pixel blanc en (0,0))",
        "Principalement blanche\n(avec 1 pixel noir en (3,5))",
        "Image aléatoire\n(simulation de bruit)"
    ],
    cols=3,
    figsize=(9, 3),
    axis=True
)
```

```
Matrice aléatoire générée :
 46 101 169  12  93  65
 21 252  17  17  80  24
224  55  61 125  71  51
 91 154 202 242 253 102
```

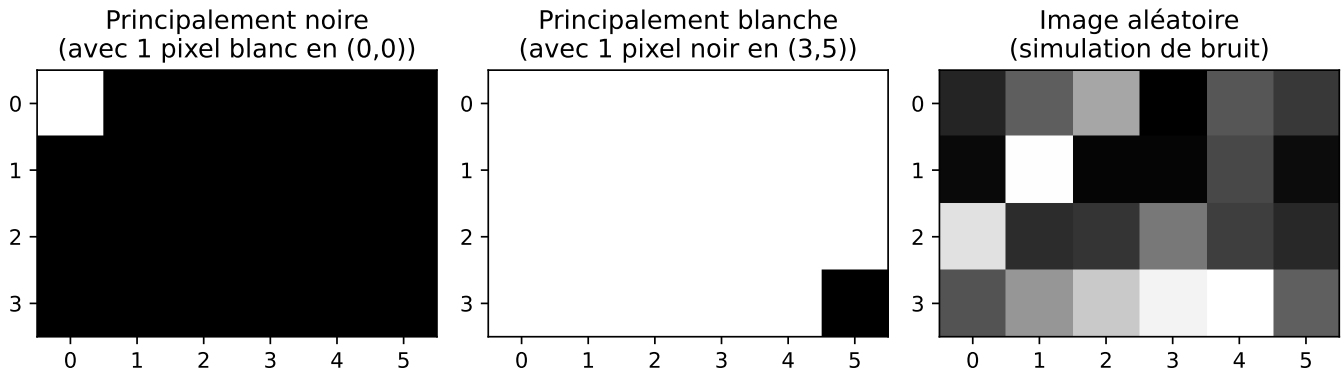


Figure 1.6: Exemples d'images synthétiques représentées matriciellement.

1.10 Lecture et Affichage d'Images

Dans des bibliothèques comme NumPy, OpenCV et scikit-image, les images numériques sont souvent représentées informatiquement sous forme de *tableaux* NumPy. Ainsi, les opérations matricielles et les concepts d'algèbre linéaire peuvent être appliqués directement au traitement d'images numériques (TIN).

L'une des opérations fondamentales en PDI est la lecture d'images.

Dans la bibliothèque `morph.py`, la fonction `mm.read()` permet de charger des images à partir de fichiers locaux ou d'URLs, comme illustré dans la Figure 2.7..

```python
import os
import numpy as np

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

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

img = np.array(img_obj)

print(f"Dimensions (H, W, Canaux) : {img.shape}")
print(f"Type de données : {img.dtype}")

mm.show(img, title="Exemple : Capture RVB")
```

```
Dimensions (H, W, Canaux) : (1365, 2048, 3)
Type de données : uint8
```

Figure 1.7: Mandrill (Mandrillus sphinx) en milieu naturel. Crédit : Julien Renoult (CC BY 4.0).

Alternative : *Téléchargement* de l'image vers un stockage local

Dans les environnements où la lecture directe d'URL est indisponible — en raison de restrictions réseau, de politiques de pare-feu ou d'absence de connectivité — une alternative consiste à effectuer au préalable le *téléchargement* de l'image vers le système de fichiers local. Après le stockage, l'image peut être chargée normalement via la fonction `mm.read()`. Dans l'exemple suivant, le fichier est enregistré localement sous le nom `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="Exemple : Capture RVB (fichier local)")
```

Explication

- `wget -O mandrill.png <URL>` effectue le *téléchargement* de l'image et la stocke localement sous le nom spécifié ;
- `mm.read('mandrill.png')` lit le fichier directement depuis le système de fichiers, sans nécessiter de requêtes HTTP supplémentaires ;
- cette stratégie réduit la dépendance à la connectivité pendant l'exécution des expériences et évite les *téléchargements* répétés de la même image.

Remarque

Le préfixe `!` est utilisé dans les environnements basés sur les *carnets* (notebooks), comme Jupyter Notebook, JupyterLab et Google Colab, pour exécuter des commandes du système d'exploitation directement dans des cellules de code. Dans les terminaux classiques, la commande doit être utilisée sans le préfixe `!`.
Si l'utilitaire `wget` n'est pas installé, on peut utiliser en alternative :

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

1.11 Conversion de Types et Seuillage

Comme nous l'avons vu, une image couleur dans l'espace RVB est représentée par la fonction :

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

C'est-à-dire que pour chaque pixel (x, y), nous avons trois valeurs (R, V, B) qui définissent sa couleur.

Conversion en Niveaux de Gris

Pour convertir une image RVB en niveaux de gris (*grayscale*), il est nécessaire de combiner les trois canaux en une seule valeur d'intensité g, qui représente la luminosité perçue. Comme l'œil humain n'est pas également sensible au rouge, au vert et au bleu, on utilise une **moyenne pondérée**. La norme ITU-R BT.601 ({ITU-R}, 2011) définit les poids suivants :

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

Après le calcul, la valeur g est arrondie à l'entier le plus proche et ajustée à l'intervalle $[0, 255]$. Le résultat est une nouvelle image, désormais en niveaux de gris, représentée par :

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

Seuillage (*Thresholding*)

À partir de l'image en niveaux de gris $f_{\text{gris}}(x, y)$, une opération fondamentale est le **seuillage**, qui produit une image **binaire** (uniquement noir et blanc). Pour cela, on choisit une valeur de coupure T (généralement dans l'intervalle $[0, 255]$) et on définit :

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

Exemple : Avec $T = 128$, les pixels dont l'intensité est supérieure à 128 deviennent blancs (255) ; les autres deviennent noirs (0).

Le seuillage est largement utilisé pour **segmenter les objets du fond**, extraire les contours ou créer des masques binaires pour un traitement ultérieur.

> **Remarque :** La valeur 255 représente le blanc maximal dans les images 8 bits, tandis que 0 représente le noir absolu.

Exemple pratique de conversion et de seuillage

La Figure 1.8 illustre les principales étapes pour transformer une image couleur en niveaux de gris, puis la convertir en image binaire par seuillage. Le code suivant implémente ces étapes :

```
# 1. Convertir en niveaux de gris
img_gray = mm.gray(img)

# 2. Appliquer un seuil (Pixels > 128 deviennent 255, autres 0)
limiar = 128
img_binaria = mm.threshold(img_gray, limiar)

# Utilisation de la nouvelle fonction
mm.show(
    [img, img_gray, img_binaria],
    titles=["Originale", "Niveaux de gris", f"Binaire (T={limiar})"],
    cols=3
)
```

Figure 1.8: Traitement de base des images : (a) image originale, (b) image en niveaux de gris, (c) image binarisée par seuil (T=128).

1.12 Limiarisation par la méthode d'Otsu

Comme présenté dans la Équation 1.4, la limiarisation convertit une image en niveaux de gris en une image binaire à l'aide d'un seuil T. Jusqu'à présent, nous avons fixé manuellement $T = 128$.

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

Cependant, le choix manuel de T n'est pas toujours trivial. La bibliothèque `mm` offre une alternative automatique : lorsque le paramètre `limiar` **n'est pas fourni**, la fonction `mm.threshold(img_gray)` calcule la valeur de T selon la **méthode d'Otsu** (OTSU, 1979). Cette méthode, qui sera détaillée dans les chapitres

suivants, maximise la variance interclasses de l'histogramme (fréquence de chaque niveau de gris), séparant automatiquement les pixels de l'objet et du fond.

Le code ci-dessous compare la limiarisation manuelle ($T = 128$) avec l'automatique (Otsu), affichant également la valeur de T calculée :

```
import cv2
# Seuillage avec T fixe (manuel)
T_fixo = 128
img_bin_fixo = mm.threshold(img_gray, T_fixo)

# Seuillage par la méthode d'Otsu (T automatique)
T_otsu, img_bin_otsu = cv2.threshold(img_gray, 0, 255,
                                     cv2.THRESH_BINARY + cv2.THRESH_OTSU)
print(f'Seuil calculé par Otsu : T = {T_otsu}')
# ou simplement :
# img_bin_otsu = mm.threshold(img_gray)

# Affichage côte à côte
mm.show(
    [img_gray, img_bin_fixo, img_bin_otsu],
    titles=["Niveaux de gris", f"Binaire (T={T_fixo})", f"Binaire (Otsu, T={T_otsu})"],
    cols=3
)
```

```
Seuil calculé par Otsu : T = 95.0
```

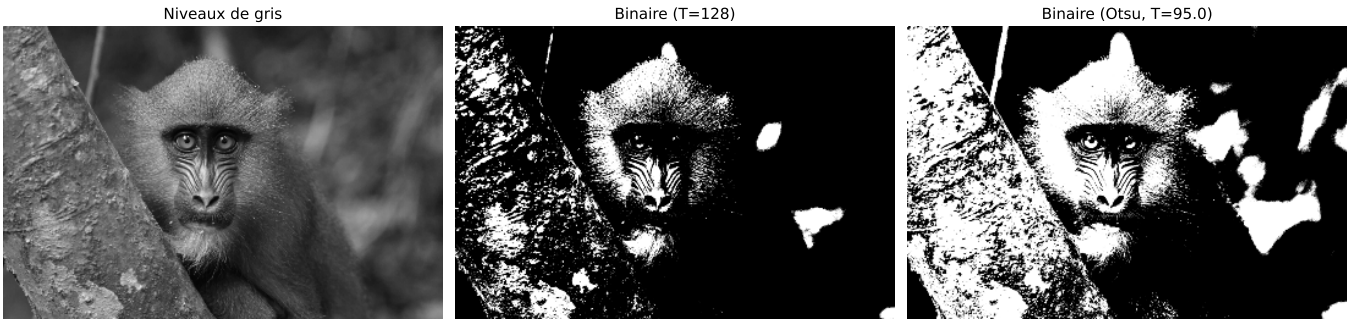


Figure 1.9: Comparaison entre seuillage manuel (T=128) et automatique (Otsu) sur l'image en niveaux de gris.

La Figure 1.9 montre que le seuil obtenu par Otsu s'adapte automatiquement à l'image, permettant une binarisation plus efficace qu'une valeur fixe, en particulier lorsque les intensités de l'objet et du fond sont bien séparées dans l'histogramme. Cette technique est largement utilisée dans les systèmes de vision par ordinateur (VC) pour la binarisation de documents, la détection d'objets et le prétraitement d'images.

Simplicité de la bibliothèque `morph.py`

Alors qu'OpenCV exige l'appel complet :

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

la bibliothèque `mm` abstrait toute cette complexité : il suffit d'appeler `mm.threshold(img_gray)`. Le seuil d'Otsu est calculé automatiquement et l'image binaire est retournée directement. Cette approche permet de se concentrer sur le concept, et non sur les détails d'implémentation.

1.13 Accès aux pixels

En Python avec NumPy, une image est structurée comme un **tableau multidimensionnel**. L'accès à un pixel spécifique utilise la convention matricielle **ligne** (axe Y) et **colonne** (axe X) : `img[ligne, colonne]`.

Le code de la Figure 1.10 démontre comment extraire ces valeurs dans des images colorées (RGB) et en niveaux de gris, ainsi que l'isolement du voisinage immédiat du point d'intérêt.

```
import matplotlib.pyplot as plt

# 1. Définition des coordonnées du pixel cible
r, c = 600, 800

# 2. Accès direct aux valeurs des pixels
pixel_cinza = img_gray[r, c]
pixel_rgb   = img[r, c]

print(f"  VALEURS AU PIXEL CIBLE ({r}, {c}):")
print(f"   • Tons de gris (Scalaire) : {pixel_cinza}")
print(f"   • Coloré (Vecteur RGB)    : R={pixel_rgb[0]}, G={pixel_rgb[1]}, B={pixel_rgb[2]}")
print("-" * 50)

# 3. Affichage didactique du voisinage 3x3 autour du pixel (r, c)
# Découpe de la ligne r-1 à r+1, et de la colonne c-1 à c+1
vizinhanca_gray = img_gray[r-1:r+2, c-1:c+2]

print(f"  MATRICE DE VOISINAGE 3x3 EN TONS DE GRIS:")
print(f"   (Le pixel cible central est mis en évidence entre crochets)\n")

# Impression formatée pour mettre en évidence le pixel 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}]") # Met en évidence le centre (100, 100)
        else:
            linhas_str.append(f" {valor:3d} ")
    print("   " + " ".join(linhas_str))

# 4. Démonstration visuelle avec mm.show (optionnel, mais excellent pour le livre)
# Affiche l'image entière avec un point délimitant la région
plt.figure(figsize=(5, 5))
plt.imshow(img)
plt.plot(c, r, 'ro', markersize=8, label=f'Pixel ({r},{c})') # Trace (x, y) -> (c, r)
plt.title(f"Localização do Pixel ({r}, {c}) na Imagem")
plt.legend()
plt.axis('on') # Garde les axes pour que l'étudiant voie les coordonnées 100, 100
plt.show()
```

```
  VALEURS AU PIXEL CIBLE (600, 800):
   • Tons de gris (Scalaire) : 81
   • Coloré (Vecteur RGB)    : R=92, G=78, B=67
--------------------------------------------------
  MATRICE DE VOISINAGE 3x3 EN TONS DE GRIS:
   (Le pixel cible central est mis en évidence entre crochets)

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

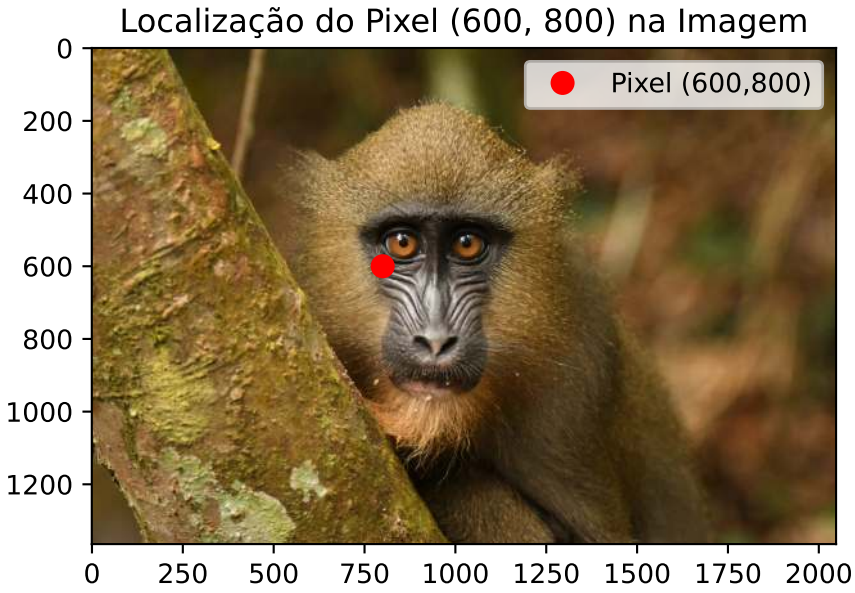


Figure 1.10: Acesso a um pixel específico (r, c) e a representação de sua vizinhança na imagem.

Explication ligne par ligne :

1. `img_gray[r, c]` — Renvoie une valeur entière unique (scalaire) comprise entre 0 et 255, représentant l'intensité de luminosité du gris.
2. `img[r, c]` — Renvoie un vecteur de trois valeurs `[R, V, B]`, correspondant aux intensités des canaux Rouge, Vert et Bleu.
3. `img_gray[r-1:r+2, c-1:c+2]` — Effectue un découpage (*slicing*) pour extraire la sous-matrice 3×3 autour du pixel. Cette opération est la base pour l'implémentation de filtres spatiaux et de convolutions.
4. La boucle `for` suivante formate simplement cette sous-matrice dans la console, affichant le pixel central entre crochets `[ ]` à des fins pédagogiques.
5. La fonction `plt.plot(c, r, 'ro')` trace un point rouge sur l'image pour corréler la coordonnée numérique avec sa position visuelle. Notez que Matplotlib utilise l'ordre d'écran standard `(X, Y)`, en inversant pour `(c, r)`.

Comme l'indexation en Python commence à zéro (*zero-based*), le coin supérieur gauche de l'image est la coordonnée `(0,0)`. Retenez toujours cette règle : la première dimension de la matrice contrôle la **hauteur** (lignes/Y) et la seconde contrôle la **largeur** (colonnes/X).

1.14 Résumé

Dans ce chapitre, nous avons présenté les fondements de la représentation des images numériques : la définition du pixel, la structuration des images en matrices et l'impact de l'échantillonnage et de la quantification sur la qualité finale :

- **Image numérique** = fonction $f(x, y)$ qui associe aux coordonnées des intensités (scalaires ou vectorielles).
- **Domaine** : ensemble fini $\mathbb{E} = \{(x, y) \in \mathbb{Z}^2 \mid 0 \leq x < L,\ 0 \leq y < H\}$.
- **Types principaux** : binaire ($\mathcal{V} = \{0, 255\}$), niveaux de gris ($\mathcal{V} = [0, 255]$) et RVB ($\mathcal{V} = [0, 255]^3$).
- **Seuillage** convertit les niveaux de gris en image binaire ; la **méthode d'Otsu** détermine automatiquement la valeur de coupure en maximisant la variance inter-classes.
- La bibliothèque `morph.py` (ou `mm`) offre des fonctions didactiques pour les opérations de base du traitement d'images, comme `mm.gray()`, `mm.threshold()`, `mm.show_multiple()`.

- **Piège NumPy** : n'utilisez jamais `[[0]*n]*m` pour créer des matrices — utilisez toujours `np.zeros()` ou `np.ones()`.
- Accès aux pixels via `img[ligne, colonne]`, avec indexation ***zero-based***.

Le chapitre 2 traitera des **histogrammes et de l'égalisation de contraste**.

1.15 🤖 Utilisation du Notebook Gemini comme tuteur complémentaire

Dans cette édition, en plus des *notebooks* interactifs sur Google Colab, le **Notebook Gemini** est disponible comme outil d'étude complémentaire. La plateforme utilise exclusivement les documents fournis par l'auteur comme base de connaissances, garantissant des réponses cohérentes avec le contenu du livre.

🎓 Étudiez avec le tuteur intelligent

Accédez à l'environnement du chapitre via le *lien* ci-dessous et explorez particulièrement les options **Guide d'étude** et **Conversation** pour approfondir votre compréhension.
🚀 ACCÉDER AU NOTEBOOK GEMINI : CHAPITRE 01

🌐 **Langue et langage de programmation**

Le projet de ce chapitre dans le Notebook Gemini a été construit uniquement avec le texte en **portugais** et les exemples de code en **Python**. Si vous étudiez avec l'édition anglaise ou française, ou si vous suivez le parcours en C++, les réponses du tuteur peuvent ne pas correspondre exactement à la version que vous lisez.

⚠️ **Avis concernant le contenu généré par l'IA**

L'IA est une alliée puissante pour les études, mais le contenu généré peut contenir des **erreurs ou des imprécisions**. Consultez toujours **des livres, des articles scientifiques et d'autres sources académiques fiables** pour valider les informations. Chaque fois que possible, exécutez les exemples pratiques fournis dans ce chapitre pour vérifier les résultats.

Fonctionnalités Disponibles sur la Plateforme

Le **Gemini Notebook** propose une suite avancée d'outils basés sur l'IA pour transformer le contenu statique du livre en une expérience d'apprentissage dynamique et multimédia. La plateforme utilise des techniques de **RAG** (*Retrieval-Augmented Generation*), fondées sur les travaux de Lewis (2020), pour ancrer les réponses strictement dans les documents fournis et minimiser l'apparition d'hallucinations.

Les principales fonctionnalités incluent :

- **Résumés Multimodaux (Audio et Vidéo)** : Génération de conversations naturelles entre experts sous forme de **Résumé Audio** (style *podcast*) et de **Résumé Vidéo**, discutant des thèmes centraux du chapitre, comme les différences entre PDI et VC, ou l'interprétation de transformations telles que le seuillage et la méthode d'Otsu.
- **Visualisation de Structures (Carte Mentale et Infographie)** : Création automatique de diagrammes connectant visuellement les concepts, par exemple, le flux de traitement depuis la capture de l'image numérique, en passant par la conversion en niveaux de gris, le seuillage et la segmentation binaire.
- **Outils d'Évaluation (Test et Cartes Pédagogiques)** : Génération de **Tests** à choix multiples et de **Cartes Pédagogiques** (*flashcards*) pour la consolidation des connaissances, basés sur le texte original (ex. : questions sur la formule de conversion RVB→niveaux de gris ou sur le fonctionnement du seuil global et de celui d'Otsu).
- **Soutien à la Présentation (*Slides* et Rapports)** : Aide à la structuration de **Présentations de *Slides*** et à la rédaction de **Rapports** techniques, facilitant la communication des résultats d'expériences

avec des images.

- **Analyse de Données (Tableau de Données)** : Organisation des données extraites du texte dans des tableaux structurés, aidant à la compréhension d'exemples pratiques, comme la comparaison entre différentes valeurs de seuil.
- ***Chat* Contextualisé** : Permet de questionner directement le code et la théorie, comme : *"Comment implémenter la conversion RVB en niveaux de gris en utilisant les pondérations de la norme UIT-R BT.601 ?"* ou *"Que devient l'image binaire si je choisis un seuil T=200 au lieu de T=128 ?"*.

1.16 Liste d'exercices

1. (15 %) Avec vos propres mots, définissez **image numérique** et **pixel**. Donnez un exemple concret de la manière dont une image colorée (RVB) est représentée sous forme matricielle dans l'ordinateur.

2. (15 %) Expliquez les différences entre **image binaire**, **niveaux de gris (8 bits)** et **image colorée RVB**, en indiquant la plage de valeurs possibles pour chaque pixel dans chaque type.

3. (20 %) En considérant la formule de conversion RVB → niveaux de gris du standard ITU-R BT.601 :

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

   Calculez la valeur du pixel en niveaux de gris pour $(R, G, B) = (80, 180, 30)$. Arrondissez à l'entier le plus proche.

4. (20 %) Qu'est-ce que la **seuillage** (*thresholding*) ? Expliquez la différence entre choisir un seuil T fixe (par exemple $T = 128$) et utiliser la **méthode d'Otsu** pour la détermination automatique du seuil. En quelques mots, comment la méthode d'Otsu choisit-elle le seuil ?

5. (15%) No contexto da biblioteca didática `mm` discutida no capítulo, responda:
   - a) (7,5%) Comment accède-t-on à la valeur du pixel à la position (ligne=50, colonne=60) d'une image en niveaux de gris `img_gray` ?
   - b) (7,5%) Quel est l'avantage d'utiliser `mm.threshold(img_gray)` sans passer le seuil ? Comparez avec l'appel équivalent dans OpenCV.

6. (15%) Que retourne la propriété `img.shape` pour une image NumPy au format RVB ? Donnez un exemple concret avec une image de 640×480 pixels.

Références du chapitre

Les fondements théoriques de ce chapitre comprennent les ouvrages suivants sur le TNI et la VC :

- Gonzalez (2018) pour les fondements du **Traitement Numérique des Images** (TNI).
- Singh (2019) pour la mise en œuvre pratique des méthodes de **traitement et d'analyse d'images**.
- Szeliski (2022) pour l'étude de la VC et des algorithmes fondamentaux.
- Bradski (2008) pour l'application de la bibliothèque **OpenCV** dans un environnement Python.
- Lewis (2020) pour le concept de **génération augmentée par récupération (RAG)**, utilisé pour le support du traitement des informations de ce matériel.

Exécuter Colab Ouvrir GitHub

1.17 Calculs de base en Python

Ce chapitre présente des exemples de codes en Python et des exercices de programmation (EPs) utilisant des ressources de base du langage, comme **l'arithmétique**, le **module math**, les **fonctions**, les **chaînes de caractères**, **les structures conditionnelles** et **itératives**.

Objectifs : savoir développer des programmes simples, en utilisant les principales structures de programmation en Python : entrées et sorties (entrées/sorties), arithmétique, fonctions, chaînes de caractères, structures conditionnelles et boucles.

ZQREF1Z

1.18 Fonction print()

La fonction `print()` est utilisée pour afficher (imprimer) du texte, des variables et le résultat de calculs à l'écran. ZQREF2Z. Voir quelques exemples :

ZQREF3Z

```
print("Bonjour le monde !")
print(3 + 4)          # affiche 7
print("Le résultat est", 3 * 2)   # affiche : Le résultat est 6
```

Exécutez les cellules ci-dessous et observez les sorties : ZQREF4Z

```
# Exemple 1 - Affichage d'une chaîne de caractères
print("Bonjour le monde !")
```

```
# Exemple 2 - Affichage d'une opération arithmétique
print(3 + 4)
```

```
# Exemple 3 - Affichage d'un texte avec une variable
print("Le résultat est", 3 * 2)
```

ZQREF5Z Utilisez la fonction `print()` avec les paramètres `sep` (séparateur) et `end` (fin), par exemple :

```
print("Python", "est", "génial", sep="-")   # sortie : Python-est-génial
print("Bonjour", end=" ")
print("le monde")          # sortie : Bonjour le monde
```

Important : Dans `print()`, plusieurs valeurs peuvent être passées, séparées par des virgules, et elles sont affichées avec un espace entre elles par défaut. ZQREF6Z.

1.19 💻 Partie Pratique avec Exercices de Programmation

🎯 Objectif de ce Cahier

Les **Exercices de Programmation (EPs)** présentés ci-dessous peuvent également être soumis dans les activités Moodle (activités VPL) qui fournissent une *rétroaction* automatique.

Ce cahier a été développé pour surmonter les limitations d'utilisation de Moodle. Avec lui, il faut :

1. **Développer :** Écrire et éditer votre solution directement dans l'environnement Colab.
2. **Valider :** Tester votre code localement en utilisant les **mêmes cas de test** que Moodle.
3. **Organiser :** Sauvegarder vos codes des activités VPL de manière sécurisée.
4. **Évaluer :** Lorsque vous êtes connecté à Moodle, il suffit de copier votre solution et de cliquer sur **Évaluer** dans Moodle (**si vous êtes sur le réseau de l'UFABC**) pour enregistrer votre note officielle.

§ Instructions Étape par Étape

Dans un environnement d'exécution (comme VSCode, Jupyter ou Colab), suivez l'ordre ci-dessous pour configurer l'environnement et valider vos exercices :

Préparation de l'Environnement

Exécutez la cellule de code ci-dessous pour télécharger `morph.py` et `testsuite.py` depuis le dépôt du cours — uniquement s'ils n'existent pas déjà dans le répertoire local. Avec les deux fichiers dans `./`, le *notebook* et les sous-processus du `TestSuite` trouvent le module sans configurations de chemin supplémentaires.

Note : Le script `testsuite.py` recherchera automatiquement les cas de test dans `all/{cap}/cases` sur GitHub.

Écriture du Code

Enregistrez votre solution dans une cellule de code en utilisant la commande magique `%%writefile`. Le nom du fichier doit suivre le modèle `EPX_Y.*`, où `X` est le chapitre, `Y` est l'exercice et `*` est l'extension du langage.

Exemple : `%%writefile EP01_01.py`

Téléchargement

Téléchargez `morph.py` et `testsuite.py` en exécutant la cellule ci-dessous :

```
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
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0 | TestSuite : 1.1.2
```

Exécution des tests

Après avoir enregistré le fichier avec votre solution, exécutez la commande ci-dessous (dans une nouvelle cellule) pour évaluer les tests automatiques :

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

Remplacez l'extension selon le langage utilisé :

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

Comment cela fonctionne : Le `TestSuite` télécharge les cas de test depuis GitHub, exécute votre programme avec chaque entrée et compare la sortie avec celle attendue – calculant automatiquement votre note.

Pour tester directement du code Python, sans enregistrer de fichier, utilisez `run_code(codigo)` en passant le code comme *chaîne de caractères* dans une variable `codigo` :

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

⚠️ Important : Règles et Bonnes Pratiques

🔹 À propos de la Saisie des Données

Votre programme doit lire l'entrée standard (clavier).

- **Python :** Utilisez `input()`.
- **Autres langages :** Utilisez la commande de lecture standard équivalente (`cin`, `Scanner`, etc.).

🔹 Configuration de l'IA dans Colab

Pour un meilleur apprentissage, il est recommandé de désactiver la complétion automatique de code par l'IA, car elle ne sera pas disponible lors des évaluations. Par exemple dans le navigateur Chrome :

- Allez dans : **Outils > Paramètres > IA générative**
- Décochez : **Activer la génération de code**

🔹 Intégrité Académique (Plagiat)

Cette ressource de tests locaux s'applique aux EPs **sans variations**. Cependant :

- **Individualité :** Chaque étudiant doit développer sa propre solution.
- **Détection de Similarité :** Le professeur utilise des outils qui détectent les copies, **même avec un changement de noms de variables ou d'espaces blancs**.

1.19.1 EP01_01 📏 Trois métriques de distance en TNI

Dans cette activité, vous devez écrire un programme qui calcule les trois distances classiques en TNI : **euclidienne (L2)**, **City-Block (L1)** et ***Chessboard* (L∞)**.

- Lisez **4 nombres réels** qui représentent les coordonnées : A_x, A_y, B_x, B_y.
- Calculez les trois distances à l'aide des formules :

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

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

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

- Affichez les trois résultats, chacun sur une ligne, formatés avec **deux décimales**, dans l'ordre : **euclidienne**, ***City-block***, ***Chessboard***.

📌 **Important** :

- Utilisez les fonctions mathématiques standard de votre langage : `math.sqrt`, `abs` (ou `fabs`) et `max`.
- La sortie doit contenir **uniquement les nombres** (un par ligne), sans texte supplémentaire.
- Voir un simulateur interactif pour cette question à la **Figure 1.11** (graphique avec glissement des points et visualisation des trois métriques).

1.19.1.1 🖼️ Pourquoi cela importe-t-il ? – Coût computationnel

Dans une image **1000×1000 pixels** (1 million de pixels), calculer la distance de chaque pixel à un point de référence exige **1 million d'opérations**. Le choix de la métrique affecte la performance :

Métrique	Opérations par pixel	Coût relatif (1M pixels)	Quand l'utiliser
Euclidienne (L2)	2 soustractions, 2 multiplications, 1 addition, 1 `sqrt`	🔴 Plus coûteuse – `sqrt` est lente	Distance « réelle » dans l'espace continu
***City-block* (L1)**	2 soustractions, 2 `abs`, 1 addition	🟡 Modérée – sans racine carrée	Grilles, robotique, images binaires
***Chessboard* (L∞)**	2 soustractions, 2 `abs`, 1 `max`	🟢 Plus efficace	Déplacements de pièces, morphologie

La fonction `sqrt` est computationnellement plus coûteuse que des opérations comme l'addition, la soustraction, la multiplication et la valeur absolue. Sur les processeurs modernes, la différence peut être faible (environ 1,5× à 3×), mais dans les systèmes embarqués ou dans des boucles de millions d'itérations, tout gain compte. Pour cette raison, **lorsque l'objectif est uniquement de comparer des distances** (ex. : trouver le point le plus proche), utilisez la **distance euclidienne au carré**.

1.19.1.2 📋 Tâche (spécification pour VPL)

Entrée :
Une seule ligne avec quatre nombres réels : `Ax Ay Bx By`

Sortie :
Trois lignes, chacune avec un nombre réel à deux décimales (euclidienne, City-block, Chessboard).

1.19.1.3 📌 Exemples

Entrée	Sortie	Observation

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

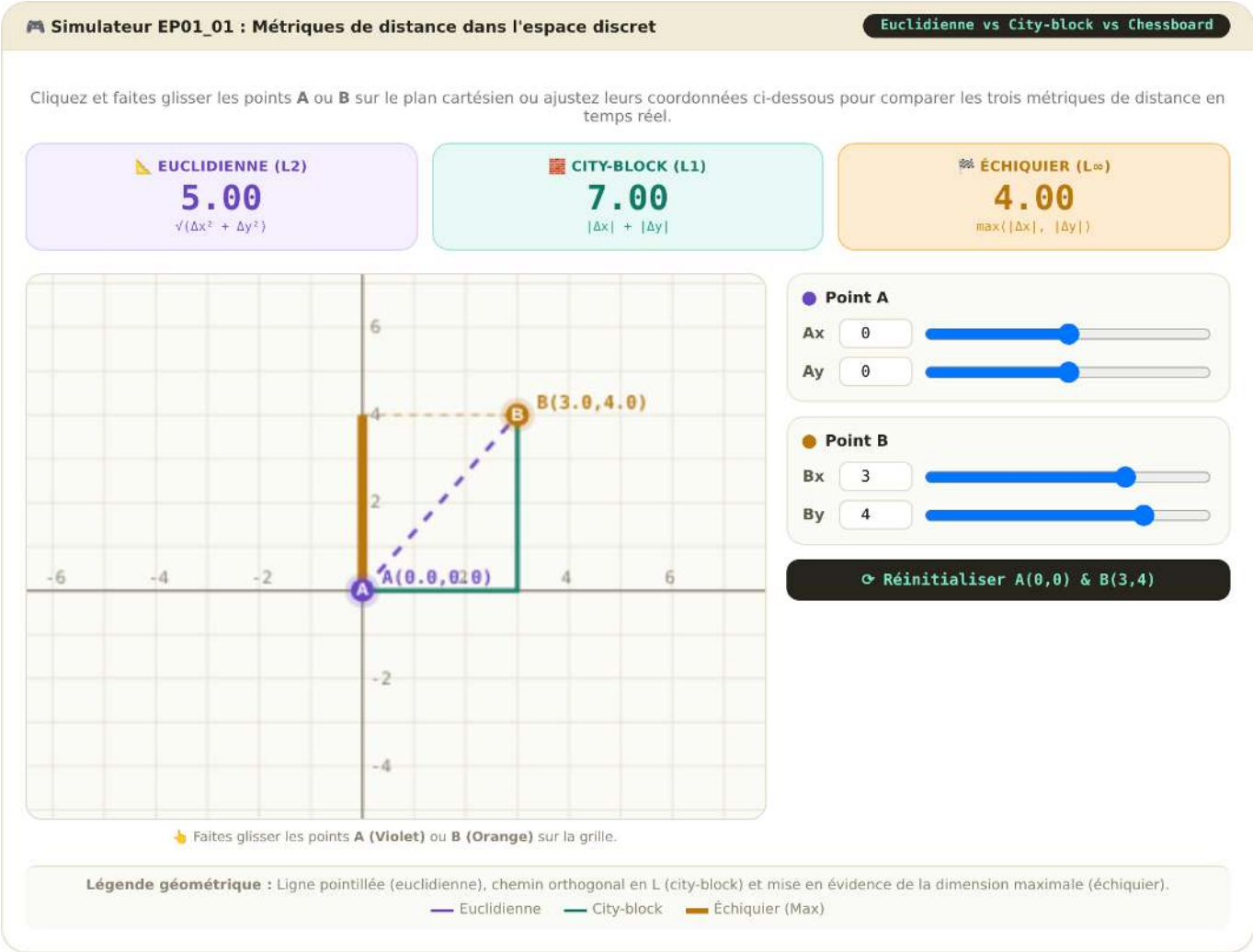
`0`
`0`
`1`
`1`
	`1.41`
	`2.00`
	`1.00`
		Diagonale unitaire

Exemple de test de `sqrt` en Python, avec `timeit` isolant chaque opération :

```python
import math
import timeit

N = 50_000_000

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

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

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

print(f"Somme simple       : {t_soma:.3f} s")
print(f"Somme + sqrt        : {t_sqrt:.3f} s")
print(f"Rapport (sqrt/somme) : {t_sqrt/t_soma:.2f}x")
```

```
Somme simple       : 2.830 s
Somme + sqrt        : 5.196 s
Rapport (sqrt/somme) : 1.84x
```



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

Figure 1.11: Simulateur EP01_01 : Distances Euclidienne, City-block et Échiquier

1.19.1.4 🐍 Python

Il suffit de créer une cellule de code normale et d'y insérer le code Python. L'entrée peut être simulée à l'aide de `input()`, qui fonctionne normalement.

Exemple de cellule :

```
%%writefile EP01_01.py
# Code Python
x1,y1,x2,y2 = int(input()), int(input()), int(input()), int(input())
# Calcul des différences
```

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

# 1. Distance euclidienne (L2)
dist_euclidiana = (dx**2 + dy**2)**0.5

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

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

# Sortie formatée selon les cas de test
print(f"{dist_euclidiana:.2f}")
print(f"{dist_city_block:.2f}")
print(f"{dist_chessboard:.2f}")
```

```
Overwriting EP01_01.py
```

```
# Attend que vous saisissiez 4 nombres entiers lors de l'exécution de cette cellule.
# Dans Jupyter ou Google Colab, la magie %run -i permet au script de lire depuis le clavier.
# Dans un terminal standard, vous utiliseriez : python3 EP01_01.py (sans le '!' et sans
'%run').

# %run -i EP01_01.py
```

```
# Envoie 4 entiers comme entrée standard (stdin) au script EP01_01.py en utilisant 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>
```

1.19.1.5 ☕ Java

Pour exécuter Java dans Colab, vous devez utiliser une cellule avec le préfixe `%%writefile` pour enregistrer le code dans un fichier, le compiler et l'exécuter.

```
%%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>
```

1.19.1.6 💻 C

De manière similaire, utilisez `%%writefile` pour enregistrer le code, puis compilez et exécutez. Pour C, nous utilisons le compilateur **GCC**.

```
# Installer le compilateur GCC et les outils de build
# Le build-essential inclut gcc, g++, make, etc.
import platform, shutil, subprocess

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

instalar_gcc()
```

```
  GCC déjà 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
```

```
# Compile le fichier .c en générant l'exécutable EP01_01
# -lm est utilisé pour lier la bibliothèque mathématique (math.h) si nécessaire

!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>
```

1.19.1.7 💻 C++

De manière similaire à Java, utilisez `%%writefile` pour enregistrer le code, puis compilez et exécutez. Rappelez-vous que, dans Colab, il est également nécessaire d'installer ce qui suit :

```
# Installer le compilateur G++ pour C++
# Le build-essential inclut g++, make, etc.
import platform, shutil, subprocess

def instalar_gpp():
    if shutil.which("g++"):
        print("  G++ déjà disponible."); return
    if platform.system() != "Linux":
        if platform.system() == "Darwin":
            print("  Mac : xcode-select --install")
        else:
            print("  Windows : utilisez WSL ou 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("  Compilateur C++ prêt.")

instalar_gpp()
```

```
  G++ déjà 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>
```

1.19.1.8 🌐 JavaScript (Node.js)

Pour JavaScript, utilisez `%%writefile` pour créer le fichier et exécutez-le avec 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>
```

1.19.1.9 📊 R

Dans Colab, on peut exécuter du code R directement en utilisant la magie `%%R`.
Le programme doit lire **quatre nombres** (x1, y1, x2, y2) et afficher la distance euclidienne avec **deux décimales**.

Exemple de cellule :

```
%%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))
```

```
# Installer R et Rscript
# Le paquet r-base installe l'environnement R complet, y compris Rscript
import platform, shutil, subprocess

def instalar_r():
    if shutil.which("R"):
        print("  R déjà 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("  Environnement R prêt.")

instalar_r()
```

```
  R déjà disponible.
```

Pour tester de la même manière que dans les exemples précédents, il faut utiliser l'entrée standard (stdin) dans le terminal ou adapter le code comme ci-dessous :

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

# Sorties : Euclidienne, City-block et 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>
```

1.19.2 EP01_02 📊 Performance Prédictive — Métriques de ML en VC

Dans cette activité, vous entrerez dans le monde de l'**Apprentissage Automatique** (*Machine Learning*). Votre objectif est d'évaluer la performance d'un classificateur binaire en calculant des métriques à partir

d'une **Matrice de Confusion**.

1.19.2.1 🧠 Pourquoi la bonne métrique est-elle importante ?

Imaginez un détecteur de **pièces de R$ 1,00**. L'impact de l'erreur définit la métrique prioritaire :

Métrique	Exemple Pratique	Importance en TDI/VC
Exactitude	**Comptage de Grains**	Utile lorsque les classes sont équilibrées (ex : la moitié des grains défectueux, l'autre moitié sains).
Précision	**Sécurité/Biométrie**	Cruciale pour éviter les **Faux Positifs** (ex : ne pas permettre à un imposteur d'accéder à un système par erreur de reconnaissance).
Sensibilité	**Santé (Tumeurs)**	Cruciale pour éviter les **Faux Négatifs** (ex : ne pas laisser une tumeur passer inaperçue lors d'un examen radiologique).
Score F1	**Billets de Banque**	Idéale pour un **équilibre** entre ne pas rejeter de vrais billets et ne pas accepter de faux billets.

1.19.2.2 📊 La Matrice de Confusion

	Prédit Positif	**Prédit Négatif**
Réel Positif	**VP** (Vrai Positif)	**FN** (Faux Négatif)
Réel Négatif	**FP** (Faux Positif)	**VN** (Vrai Négatif)

Tâche :

1. Lisez **4 valeurs entières** dans l'ordre : `VP`, `FN`, `FP`, `VN`.
2. Calculez les métriques à l'aide des formules :

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

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

$$\text{Sensibilité (Rappel)} = \frac{VP}{VP + FN}$$

$$\text{Score F1} = \frac{2 \times \text{Précision} \times \text{Sensibilité}}{\text{Précision} + \text{Sensibilité}}$$

3. Affichez les résultats formatés avec **deux décimales**, un par ligne.

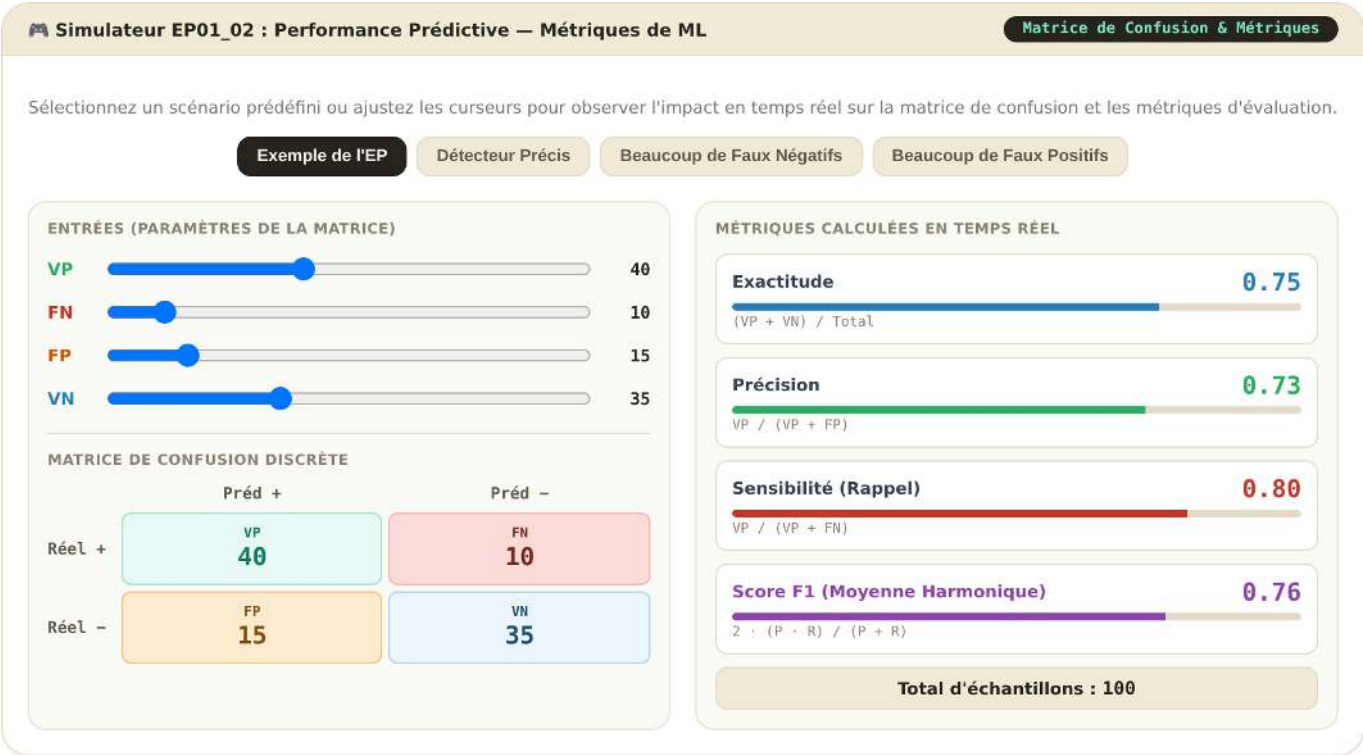
📌 **Important** :

- Utilisez la division en virgule flottante pour éviter des résultats tronqués.
- L'ordre de sortie doit être : **Exactitude**, **Précision**, **Sensibilité** et **Score F1**.
- Consultez la simulation interactive de cet EP sur la Figure 1.12.

1.19.2.3 📌 Exemple d'Exécution

Entrée	Sortie	Observation
40	0.75	Exactitude
10	0.73	Précision
15	0.80	Sensibilité
35	0.76	Score F1

(Note : VP=40, FN=10, FP=15, VN=35. Nombre total de cas = 100)



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

Figure 1.12: Simulateur EP01_02 : Performance Prédictive — Métriques de ML

```
%%writefile EP01_02.py
# votre solution
```

```
Overwriting EP01_02.py
```

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

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

1.19.3 EP01_03 ↗ *Mean Average Precision* (mAP) — Courbe Précision-Sensibilité

Dans cette activité, vous évaluerez un classificateur binaire (ex. : détection de déforestation dans des images satellites, voir dgi.inpe.br) à travers la **courbe Précision-Sensibilité** et la métrique **mAP (*Mean Average Precision*)**. Le mAP est standard dans des compétitions comme COCO (*Common Objects in Context*) et PASCAL VOC (*Visual Object Classes*) et des modèles YOLO (*You Only Look Once*).

1.19.3.1 🧠 Pourquoi le mAP est-il la métrique-standard ?

Dans l'EP01_02, vous avez vu que le choix du seuil modifie significativement la Précision et la Sensibilité. Le **mAP (*Mean Average Precision*)** résout cela : il évalue le modèle à **plusieurs seuils** (chaque seuil doit générer une matrice de confusion différente) et résume la performance par **l'aire sous la courbe Précision-Sensibilité (P-S)**.

Alors que le F1-*Score* examine un unique point d'équilibre, le mAP considère la courbe entière. Plus il est proche de **1,0**, meilleur est le détecteur à tous les seuils et classes (ex. : pièces de 25, 50 et 1 réal).

Métrique	Ce qu'elle résume	Limitation
F1-*Score*	Équilibre P × S à un seuil unique	Dépend du seuil choisi
AP	Aire sous la courbe P-S d'une clase	Valide uniquement pour une classe
mAP	Moyenne des AP sur toutes les classes	Plus complexe à implémenter

Références : Roboflow — mAP · Vidéo explicative

1.19.3.2 Comment le mAP est-il calculé — pas à pas

1. **Seuils fixes** (utilisez toujours cette liste) :

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

2. Pour chaque seuil (t), classez les échantillons : `predito = 1 se confiança  t, senão 0`. Calculez VP, FP, FN, VN et obtenez Précision((t)) et Sensibilité((t)).
3. Construisez la courbe P-S : paires (Sensibilité((t)), Précision((t))), triées par Sensibilité croissante.
4. **Monotonisez la Précision** :

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

5. Calculez l'**AP** (aire sous la courbe monotone) en utilisant la **règle du trapèze** (approximation plus précise que la simple somme 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** = moyenne des AP de toutes les classes. Dans cet EP, il n'y a qu'**1 classe**, donc mAP = AP.

> **Note**
>
> 📐 **Différence résumée :**
> La *somme de Riemann* approxime l'aire par des rectangles, pouvant sous-estimer ou surestimer. La **règle du trapèze** utilise des trapèzes, réduisant l'erreur en considérant la moyenne entre les valeurs aux extrémités de l'intervalle, étant généralement plus précise pour des fonctions lisses par morceaux, comme la courbe Précision-Sensibilité.

1.19.3.3 📋 Tâche

Lisez un entier `n` (quantité d'échantillons). Ensuite, lisez `n` lignes, chacune avec : `verdade` (0 ou 1) et `confiança` (float 0.0–1.0).

Calculez et imprimez, pour le **seuil 0.85** (index 9 de la liste) :

- Matrice de Confusion (VP, FN, FP, VN)

- Acurácia, Précision, Sensibilité et F1-Score

Ensuite, pour **tous les seuils**, imprimez :

- Précisions brutes, Précisions monotones et Sensibilités, séparées par **,**
- mAP final

1.19.3.4 📌 Important

- Seuil fixe pour les métriques individuelles : **0.85**
- Division sécurisée : si le dénominateur est nul, utilisez 0
- Formatage : deux décimales
- Monotonisez de la fin vers le début
- La Figure 1.13 présente une simulation de cette question

1.19.3.5 📌 Exemple d'Exécution

Entrée	Sortie Attendue
7 0 0.94 1 0.80 1 0.69 0 0.67 1 0.30 1 0.15 1 0.15	# MÉTRIQUES POUR LE SEUIL 0.85 # Matrice de Confusion : VP = 0, FN = 5 FP = 1, VN = 1 Métriques d'Évaluation : Acurácia : 0.14 Précision : 0.00 Sensibilité : 0.00 F1-Score : 0.00 # MÉTRIQUES POUR TOUS LES SEUILS # Précisions : 0.00, 0.00, 0.00, 0.50, 0.50, 0.50, 0.50, 0.50, 0.60, 0.71, 0.71 Précisions mon. : 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71, 0.71 Sensibilités : 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

1.19.3.6 🐍 Astuce pour calculer l'AP (avec la règle du trapèze)

```python
def calcular_AP(verdades, confiancas, limiares):
    m = len(limiares)
    precisoes = [0.0] * m
    sensibilidades = [0.0] * m
    for i in range(m):
        p, s = calcular_metricas(verdades, confiancas, limiares[i])
        precisoes[m-1-i] = p
        sensibilidades[m-1-i] = s
    prec_mono = precisoes.copy()
    for i in range(m-2, -1, -1):
        if prec_mono[i] < prec_mono[i+1]:
            prec_mono[i] = prec_mono[i+1]
    AP = 0.0
    for i in range(1, m):
        # Règle du trapèze : moyenne des hauteurs multipliée par la base
        area_trapezio = (prec_mono[i-1] + prec_mono[i]) / 2.0
```

```
        AP += area_trapezio * (sensibilidades[i] - sensibilidades[i-1])
    return precisoes, prec_mono, sensibilidades, AP
```

```
%%writefile EP01_03.py
# votre solution
```

```
Overwriting EP01_03.py
```

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

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

1.19.4 EP01_04 🖼️ Lecture et Informations d'une Image sous forme de Matrice

Dans cette activité, vous devez écrire un programme qui traite une image numérique représentée comme une matrice de pixels en niveaux de gris.

- Lisez deux entiers **L** et **C**, représentant le nombre de lignes et de colonnes.
- Lisez les **L * C** valeurs entières qui composent la matrice de l'image (chaque valeur entre 0 et 255).
- Calculez et imprimez les informations suivantes :

1. Le nombre de lignes.
2. Le nombre de colonnes.
3. La valeur du pixel le plus grand (**Maximum**).
4. La valeur du pixel le plus petit (**Minimum**).
5. La **Moyenne** arithmétique de tous les pixels.

📌 **Important** :

- La sortie doit suivre exactement le format étiqueté (ex : `Linhas: X`).
- La valeur de la moyenne doit être formatée avec **deux décimales**.
- Voir un simulateur interactif pour cette question dans **Figure 1.14** (grille interactive pour visualiser les intensités et effectuer les calculs en temps réel).

1.19.4.1 🧠 Pourquoi cela importe-t-il ? – L'Image comme Données

Toute image numérique est, au fond, une structure de données. En niveaux de gris 8 bits, chaque pixel est une valeur scalaire. Extraire des statistiques de base est la première étape pour :

Opération	Utilité Pratique
Maximum/Minimum	Identifier si l'image est « délavée » (faible contraste) ou saturée.
Moyenne	Calculer la luminosité globale de la scène pour des ajustements d'exposition.
Normalisation	Redimensionner les valeurs vers des intervalles comme $[0, 1]$ dans les réseaux de neurones.

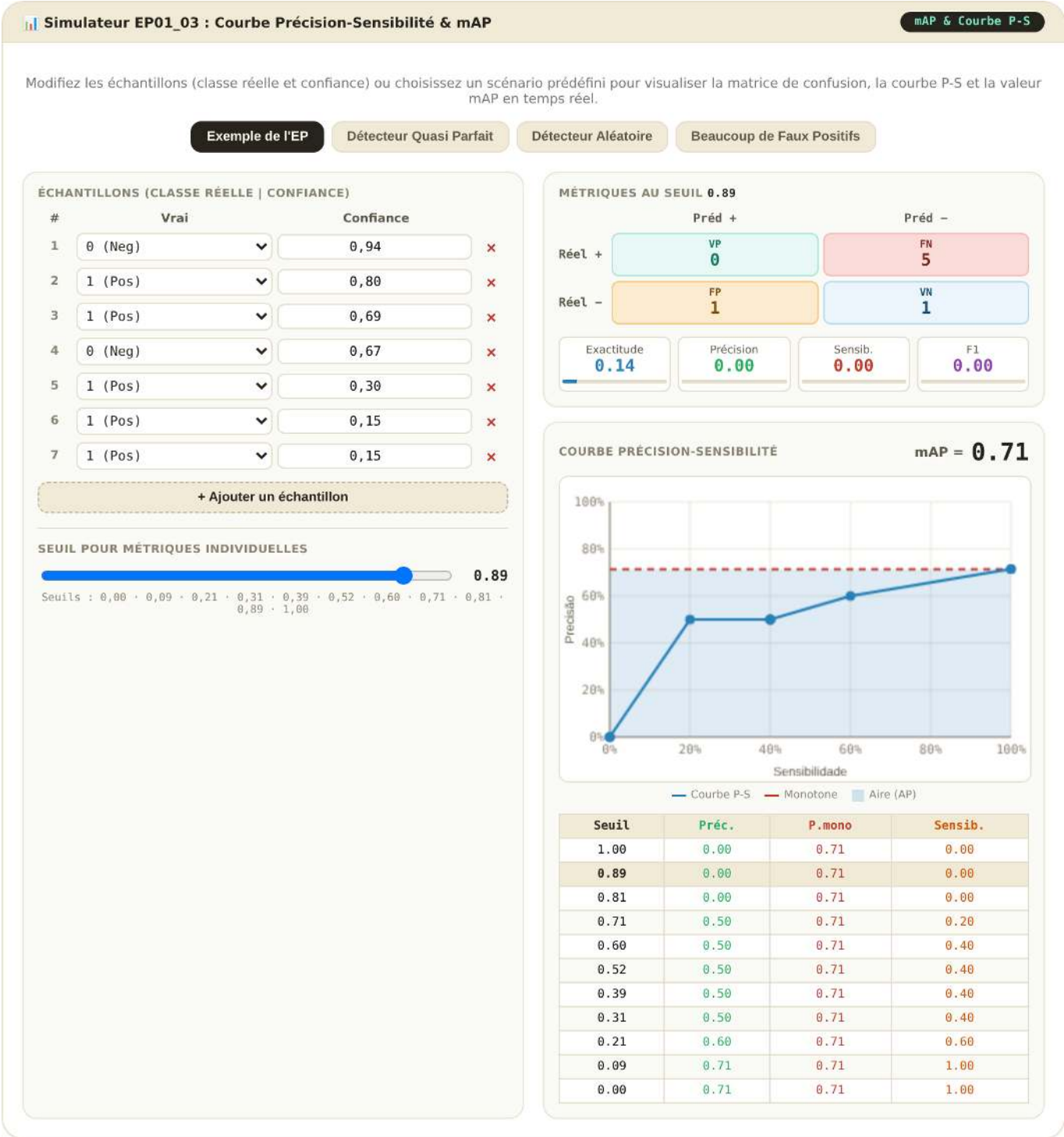
1.19.4.2 📋 Tâche (spécification pour VPL)

Entrée :

La première ligne contient l'entier `L` (lignes).

La deuxième ligne contient l'entier `C` (colonnes).

Les lignes suivantes contiennent les éléments de la matrice.



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

Figure 1.13: Simulateur EP01_03 : Mean Average Precision (mAP) et courbe P-S

Sortie :

Cinq lignes formatées selon l'exemple :

```
Linhas: L
Colunas: C
Max: V
Min: V
Media: V.VV
```

1.19.4.3 📌 Exemples

Entrée	Sortie	Observation
2 3 0 128 255 50 100 200	Linhas: 2 Colunas: 3 Max: 255 Min: 0 Media: 122.17	Petite image à fort contraste

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

Figure 1.14: Simulateur EP01_04 : Statistiques de pixels dans une matrice discrète 5x5

```
%%writefile EP01_04.py
# votre solution
```

```
Overwriting EP01_04.py
```

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

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

1.19.5 EP01_05 🔄 Négatif d'une Image en Niveaux de Gris

Dans cette activité, il faut écrire un programme qui calcule le négatif d'une image numérique.

- Lire deux entiers **L** et **C**, représentant le nombre de lignes et de colonnes.
- Lire les valeurs entières qui composent la matrice de l'image.
- Pour chaque pixel, appliquer la transformation d'inversion :

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

- Afficher la matrice résultante, en conservant le format original (L lignes et C colonnes).

📌 **Important** :

- Les valeurs de chaque ligne dans la sortie doivent être séparées par un **espace blanc**.
- La sortie doit contenir **uniquement les nombres** de la matrice résultante.
- Voir un simulateur interactif pour cette question à la **Figure 1.15** (comparaison en temps réel entre la matrice originale et son négatif).

1.19.5.1 🧠 Pourquoi cela importe-t-il ? – Inversion d'Intensité

Le **négatif** est une transformation linéaire de base qui inverse l'échelle de luminosité. C'est un outil essentiel pour que l'œil humain identifie les détails clairs qui sont « cachés » dans des fonds plus sombres, et il est largement utilisé dans :

Application	Utilité
Imagerie Médicale	Améliore la visualisation des anomalies dans les tissus denses (ex : Rayons X).
Astronomie	Mettre en évidence les galaxies et les nébuleuses ténues contre le vide de l'espace.
Arts Numériques	Effets esthétiques et préparation de masques de sélection.

1.19.5.2 📋 Tâche (spécification pour VPL)

Entrée :

La première ligne contient l'entier `L`.

La deuxième ligne contient l'entier `C`.

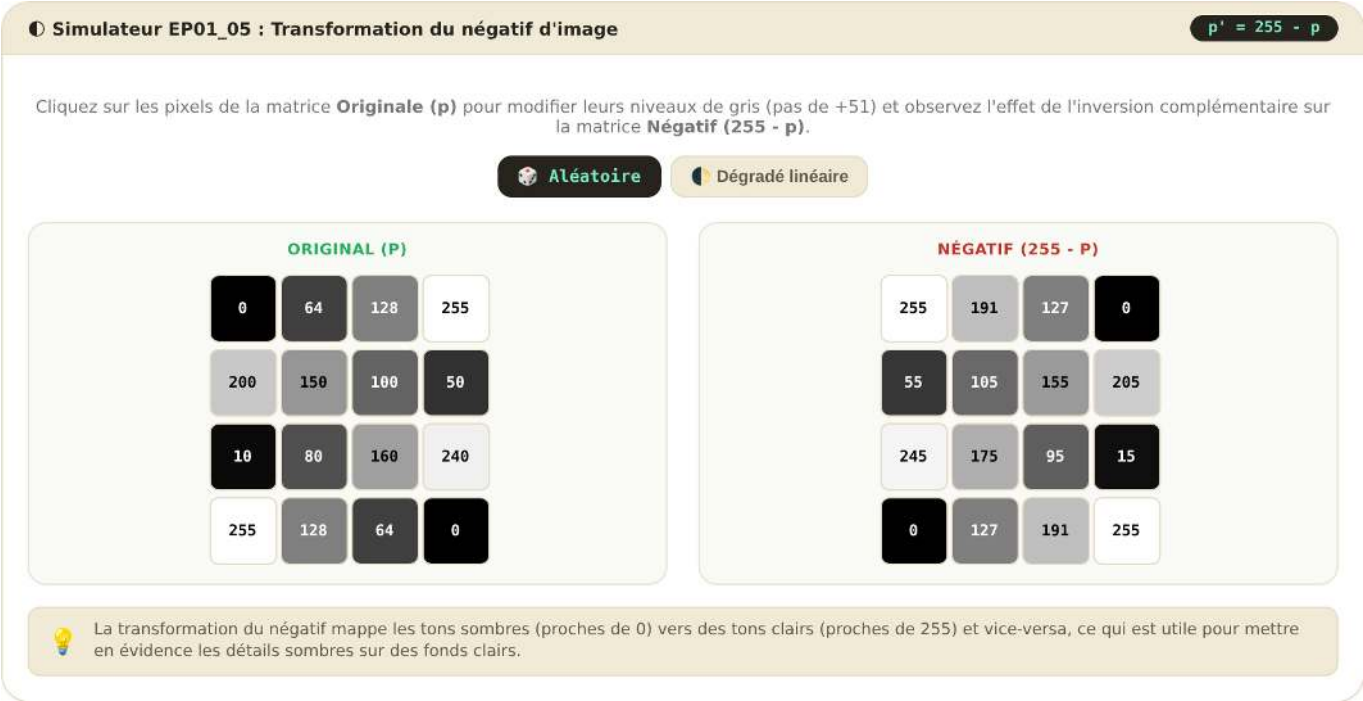
Les lignes suivantes contiennent les éléments de la matrice.

Sortie :

La matrice inversée avec `L` lignes et `C` colonnes.

1.19.5.3 📌 Exemples

Entrée	Sortie	Remarque
2 3 0 128 255 50 100 255	255 127 0 205 155 55	Là où il y avait 0 (noir), devient 255 (blanc)



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

Figure 1.15: Simulateur EP01_05 : Transformation de Négatif d'Image (Inversion d'Intensité)

```
%%writefile EP01_05.py
# votre solution
```

```
Overwriting EP01_05.py
```

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

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

1.19.6 EP01_06 🎨 Conversion RGB → Niveaux de gris (ITU-R BT.601)

Dans cette activité, vous devez écrire un programme qui convertit des pixels colorés (RVB) en niveaux de gris en utilisant la pondération physiologique de la norme ITU-R BT.601.

- Lisez deux entiers **L** et **C**, représentant le nombre de lignes et de colonnes.
- Lisez **L** × **C** triplets d'entiers, où chaque triplet représente les canaux **R** (Rouge), **G** (Vert) et **B** (Bleu) d'un pixel.
- Pour chaque pixel, calculez la valeur de gris (g) à l'aide de la formule :

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

- Affichez la matrice résultante (L lignes et C colonnes) contenant les valeurs entières converties.

📌 **Important** :

- Utilisez la fonction `round()` de votre langage pour garantir l'arrondi correct à l'entier le plus proche.
- La sortie doit contenir uniquement les valeurs de gris, en préservant la structure de la matrice (séparées par des espaces sur la ligne).
- Consultez un simulateur interactif pour cette question sur la **Figure 1.16** (ajustez les curseurs pour voir comment chaque couleur contribue à la luminosité finale).

1.19.6.1 🧠 Pourquoi ne pas utiliser simplement la moyenne ?

L'œil humain ne perçoit pas toutes les couleurs avec la même intensité. Nous sommes beaucoup plus sensibles au **Vert** qu'au **Bleu** en raison de notre évolution biologique. La norme ITU-R BT.601 utilise des poids spécifiques pour créer une image en niveaux de gris qui semble naturellement correcte pour notre vision :

Canal	Poids	Perception humaine
🟢 **Vert**	**58,7 %**	Sensibilité maximale (distinction du feuillage).
🔴 **Rouge**	**29,9 %**	Sensibilité moyenne.
🔵 **Bleu**	**11,4 %**	Faible sensibilité (tons plus sombres).

1.19.6.2 📋 Tâche (spécification pour VPL)

Entrée :

La première ligne contient l'entier `L`.

La deuxième ligne contient l'entier `C`.

Les lignes suivantes contiennent des triplets d'entiers `R G B` pour chaque pixel.

Sortie :

La matrice des niveaux de gris avec `L` lignes et `C` colonnes.

1.19.6.3 📌 Exemples

Entrée	Sortie	Observation
1 3 255 0 0 0 255 0 0 0 255	76 150 29	Remarquez comment le Vert (150) est plus lumineux que le Bleu (29)

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

Figure 1.16: Simulador EP01_06 : Conversion RVB en niveaux de gris (pondération perceptuelle UIT-R BT.601)

```
%%writefile EP01_06.py
# votre solution
```

```
Overwriting EP01_06.py
```

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

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

1.19.7 EP01_07 ⚫ Seuillage Manuel : Image Binaire

Dans cette activité, vous devez écrire un programme qui réalise la segmentation d'une image par seuillage (*thresholding*).

- Lisez deux entiers **L** et **C**, représentant les dimensions de la matrice.
- Lisez un entier **T**, qui sera la valeur du seuil (coupure).
- Lisez les valeurs entières de la matrice.
- Pour chaque pixel p, appliquez la règle de binarisation suivante :

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

- Affichez la matrice résultante contenant uniquement les valeurs 0 ou 255.

📌 **Important** :

- Faites attention à l'opérateur : le pixel ne devient blanc (255) que s'il est **strictement supérieur** à T.
- La sortie doit conserver la structure matricielle (L lignes et C colonnes).
- Voir un simulateur interactif pour cette question sur la **Figure 1.17** (ajustez le curseur de T pour observer comment les objets sont isolés du fond).

1.19.7.1 🧠 Qu'est-ce que la Segmentation ?

Le seuillage est la méthode la plus simple pour séparer les objets d'intérêt du fond de l'image. En transformant les niveaux de gris en noir et blanc pur, on crée une carte binaire qui facilite le comptage des objets ou l'identification des formes :

Valeur du Pixel (p)	Condition	Résultat Final
Sombre ($p \leq T$)	Fond/Bruit	**0** (Noir)
Clair ($p > T$)	Objet/Mise en évidence	**255** (Blanc)

1.19.7.2 📋 Tâche (spécification pour VPL)

Entrée :

La première ligne contient l'entier `L`.

La deuxième ligne contient l'entier `C`.

La troisième ligne contient l'entier `T` (seuil).

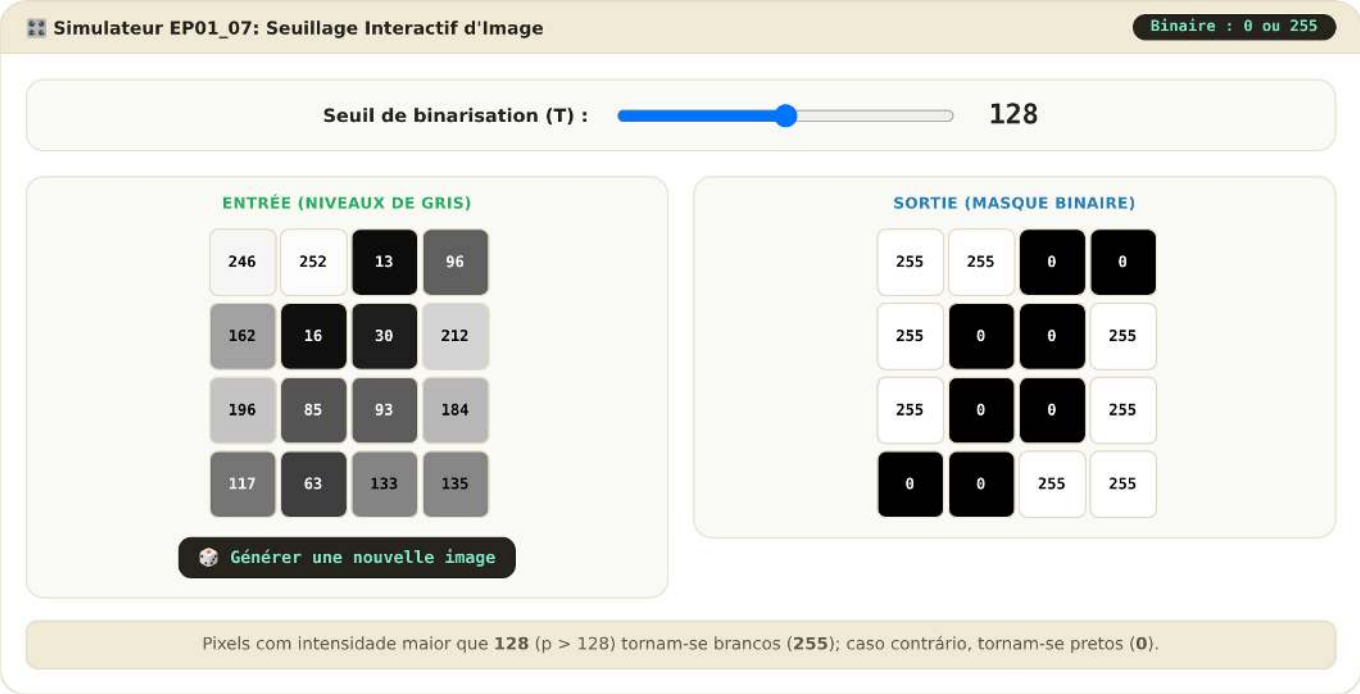
Les lignes suivantes contiennent les éléments de la matrice.

Sortie :

La matrice binarisée (0 ou 255) avec `L` lignes et `C` colonnes.

1.19.7.3 📌 Exemples

Entrée	Sortie	Observation
2 4 128 0 100 128 200 50 129 255 64	0 0 0 255 0 255 255 0	Notez que la valeur 128 est devenue 0 (car 128 $\leq$ 128)



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

Figure 1.17: Simulateur EP01_07 : Seuillage global interactif (binarisation p > T)

```
%%writefile EP01_07.py
# votre solution
```

```
Overwriting EP01_07.py
```

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

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

1.19.8 EP01_08 🎨 Remappage par Plage d'Intensité

Dans cette activité, vous devez écrire un programme qui applique des transformations linéaires distinctes à différentes régions d'intensité de l'image.

- Lisez deux entiers **L** et **C**, représentant les dimensions de la matrice.
- Lisez les entiers **T** (seuil), (delta 1) et (delta 2).
- Lisez les valeurs entières de la matrice.
- Pour chaque pixel p, appliquez la règle de remappage conditionnel :

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

- Affichez la matrice résultante avec les nouvelles valeurs d'intensité.

📌 **Important** :

- est le décalage appliqué aux pixels sombres (en dessous du seuil).
- est le décalage appliqué aux pixels clairs (supérieurs ou égaux au seuil).
- Les cas de test garantissent que le résultat sera toujours dans l'intervalle valide de **0 à 255**, il n'est donc pas nécessaire de gérer la saturation ou les arrondis.
- La sortie doit conserver la structure de la matrice (**L** lignes et **C** colonnes).

1.19.8.1 🧠 Transformation Conditionnelle des Pixels

En Traitement Numérique des Images (TNI), nous devons souvent traiter les régions de manière indépendante. Cette technique permet, par exemple, d'éclaircir uniquement les ombres d'une photographie (en augmentant les pixels sombres) sans saturer la luminosité des zones déjà claires, ou vice versa.

Plage d'Intensité	Condition	Opération
Pixels Sombres	$p < T$	$p + \delta_1$
Pixels Clairs	$p \geq T$	$p + \delta_2$

1.19.8.2 📋 Tâche (spécification pour VPL)

Entrée :

La première ligne contient les entiers `L` et `C`.

La deuxième ligne contient les entiers `T`, et .

Les lignes suivantes contiennent les éléments de la matrice.

Sortie :

La matrice transformée avec `L` lignes et `C` colonnes, avec des valeurs séparées par des espaces.

1.19.8.3 📌 Exemples

Entrée	Sortie	Observation

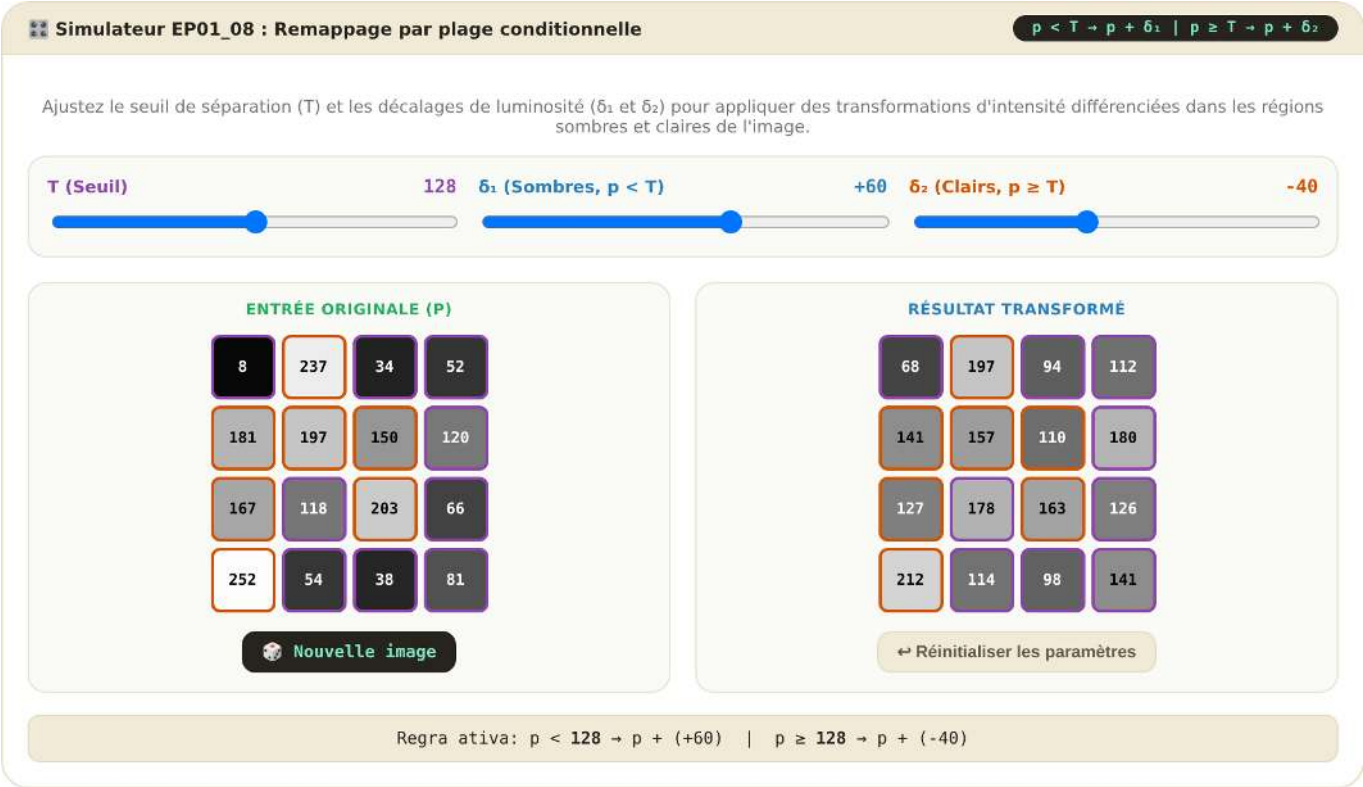
`2 4`
`128 60 -40`
`0 100 150 255`
`80 128 200 30`
	`60 160 110 215`
	`140 88 160 90`
		Les pixels < 128 ajoutent 60. Les pixels 128 soustraient 40.

```
%%writefile EP01_08.py
# votre solution
```

```
Overwriting EP01_08.py
```

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

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



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

Figure 1.18: Simulateur EP01_08 : Remappage par Plage Conditionnelle (Luminosité et Contraste par Seuil)

1.19.9 EP01_09 🏁 Modèle de Damier : Matrice d'Échecs

Dans cette activité, vous devez écrire un programme qui génère une image synthétique selon le motif d'un damier.

- Lisez deux entiers **L** (lignes) et **C** (colonnes).
- Générez une matrice où les valeurs alternent entre **0** (noir) et **1** (blanc).
- La logique de remplissage doit suivre la règle de parité :
- L'élément à la position $(0, 0)$ est toujours **0**.
- Un pixel à la position (i, j) sera **1** si la somme des indices $(i + j)$ est **impaire**.
- Un pixel à la position (i, j) sera **0** si la somme des indices $(i + j)$ est **paire**.

📌 **Important** :

- Les couleurs doivent alterner correctement aussi bien horizontalement que verticalement.
- La sortie doit être la matrice imprimée ligne par ligne, avec les éléments séparés par un espace.
- Voir un simulateur interactif pour cette question sur la **Figure 1.19** (ajustez les dimensions pour visualiser la construction de la grille et la sortie textuelle correspondante).

1.19.9.1 🧠 Motifs Synthétiques

Créer des motifs géométriques est un exercice fondamental pour maîtriser la **logique des indices** dans les matrices. En Traitement Numérique des Images, le motif du damier n'est pas seulement esthétique ; il est largement utilisé pour :

Application	Utilité
Calibrage de Caméra	Estimer les paramètres intrinsèques et extrinsèques de l'objectif.
Correction de Distorsion	Identifier et corriger l'effet de "barillet" ou "coussinet" sur les objectifs grand-angle.

Application	Utilité
Cartographie 3D	Projeter des motifs connus pour reconstruire des surfaces dans les systèmes de lumière structurée.

1.19.9.2 📋 Tâche (spécification pour VPL)

Entrée :

Une ligne contenant l'entier `L` (lignes).

Une ligne contenant l'entier `C` (colonnes).

Sortie :

La matrice d'échecs avec `L` lignes et `C` colonnes, imprimée avec des espaces entre les éléments.

1.19.9.3 📌 Exemples

Entrée	Sortie	Observation
3 4	0 1 0 1 1 0 1 0 0 1 0 1	Notez que chaque ligne commence par l'inverse de la précédente

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

Figure 1.19: Simulateur EP01_09 : Générateur de motif en damier (logique de parité (i + j) % 2)

```
%%writefile EP01_09.py
# votre solution
```

```
Overwriting EP01_09.py
```

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

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

1.19.10 EP01_10 📄 Métadonnées : Lecture de fichier PGM

Dans cette activité, vous devez lire un fichier image au format **PGM (Portable Gray Map)** et extraire ses dimensions à partir de l'en-tête.

- Le format **PGM (P2)** est un fichier texte simple (ASCII) qui stocke des images en niveaux de gris.
- Le fichier possède un **en-tête** structuré de la manière suivante :

1. **Version :** L'identifiant `P2`.
2. **Commentaires :** Lignes optionnelles commençant par `#` (à ignorer).
3. **Dimensions :** Deux entiers représentant la **Largeur** et la **Hauteur**.
4. **Maximum :** Un entier représentant l'intensité maximale (généralement 255).

- Après l'en-tête, suivent les données des pixels.

📌 **Important** :

- **Lecture du fichier :** Vous devez ouvrir le fichier indiqué dans l'exemple en utilisant la fonction `open()` de Python.
- **Ordre de sortie :** Contrairement à l'ordre présent dans le fichier, la sortie attendue doit être au format de tuple : `(Hauteur, Largeur, Canaux)`.
- Comme les fichiers PGM sont en niveaux de gris, le nombre de **Canaux** est toujours **1**.
- Consultez un simulateur interactif pour cette question sur **Figure 1.20** (ajustez les dimensions pour voir comment l'en-tête ASCII est généré).

1.19.10.1 🧠 Comprendre le format PGM

Le format PGM est l'un des plus simples pour le traitement d'images. Étant en texte pur, il permet de visualiser les métadonnées et même les valeurs des pixels en ouvrant le fichier dans un bloc-notes :

Composant	Exemple	Signification
Nombre magique	`P2`	Identifie qu'il s'agit d'un PGM au format texte (ASCII).
Commentaire	`# CREATOR...`	Ligne informative ignorée par le processeur.
Dimensions	`397 343`	**397** colonnes (Largeur) et **343** lignes (Hauteur).
Intensité	`255`	Définit la valeur du blanc pur (échelle de 0 à 255).

1.19.10.2 📋 Tâche (spécification pour VPL)

Entrée :

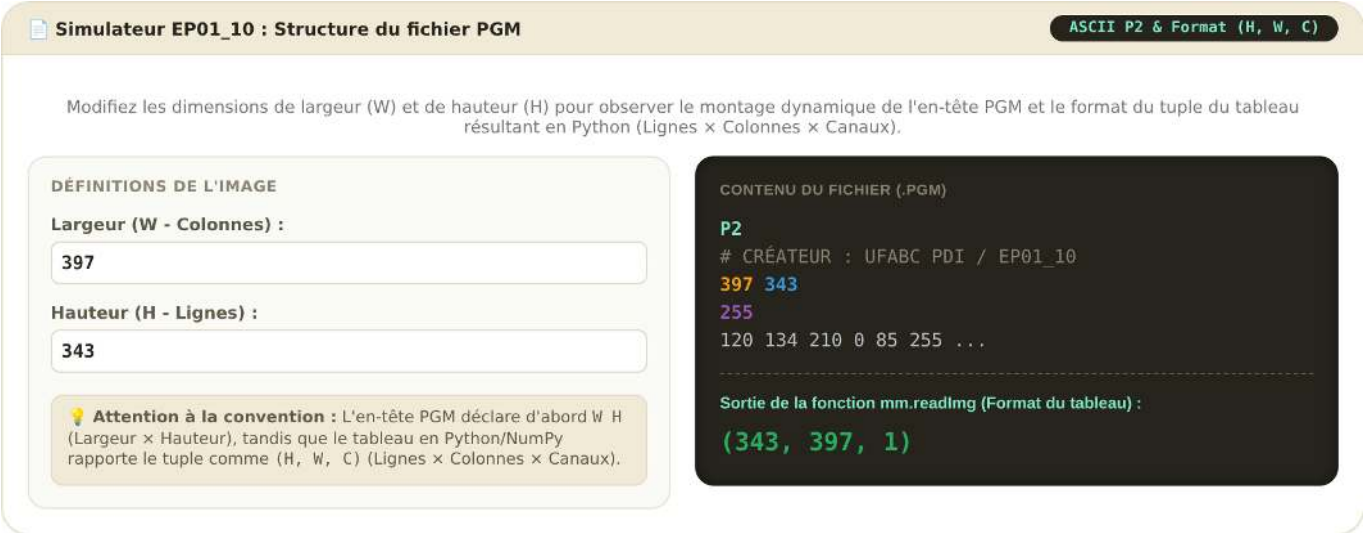
Aucune entrée via le clavier. Le programme doit lire le fichier `"aula01fig03b.pgm"` présent dans le répertoire d'exécution.

Sortie :

Un tuple contenant `(Hauteur, Largeur, 1)`.

1.19.10.3 📌 Exemples

Nom du fichier	Sortie attendue	Remarque
"aula01fig03b.pgm"	(343, 397, 1)	Notez l'inversion de l'ordre : la Hauteur en premier



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

Figure 1.20: Simulateur EP01_10 : Structure du fichier PGM (En-tête ASCII P2 et mappage vers un tuple Python)

> **Note**
>
> Le fichier nécessaire pour ce TP sera téléchargé automatiquement depuis le dépôt via le code suivant :

```
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"  Fichier téléchargé : {f}")
        except urllib.error.HTTPError as e:
            print(f"  Erreur {e.code} : Introuvable dans {url}")
```

```
%%writefile EP01_10.py
# votre solution
```

```
Overwriting EP01_10.py
```

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

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

1.19.11 EP01_11 ↗ Analyse de voisinage : Filtre de maximum 1D

Dans cette activité, vous devez implémenter un filtre morphologique simple de maximum opérant sur un signal unidimensionnel (vecteur).

- Lisez un entier **n**, représentant la taille du vecteur.
- Lisez les **n** éléments entiers qui composent le vecteur original **v1**.
- Créez un nouveau vecteur **v2**, où chaque position i est le résultat de la comparaison entre l'élément courant et ses voisins immédiats :

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

📌 **Important** :

- **Bords :** Aux extrémités du vecteur (indices 0 et $n-1$), le voisinage ne contient que deux éléments (l'élément lui-même et le seul voisin disponible). À l'indice 0, comparez uniquement $v1[0]$ et $v1[1]$. Au dernier indice, comparez uniquement $v1[n-2]$ et $v1[n-1]$.
- **Sortie :** Affichez l'en-tête « v2 : » suivi des valeurs du vecteur résultant, une par ligne.
- Consultez un simulateur interactif pour cette question dans la **Figure 1.21** (survolez les résultats avec la souris pour visualiser la fenêtre de voisinage utilisée dans le calcul).

1.19.11.1 🧠 Pourquoi analyser les voisins ?

En traitement d'images, la valeur d'un pixel est rarement isolée ; elle dépend du contexte qui l'entoure. Le **Filtre de Maximum** est la base de l'opération de **Dilatation** en morphologie mathématique, servant à :

Fonction	Effet visuel
Rehaussement	Étend les structures brillantes et « épaissit » les objets clairs.
Suppression du bruit	Élimine les petits points noirs (bruit « sel et poivre » sombre).
Remplissage	Ferme les petits trous ou lacunes dans les formes binaires.

1.19.11.2 📋 Tâche (spécification pour VPL)

Entrée :

Un entier `n`.

Sur les lignes suivantes, les `n` éléments entiers du vecteur.

Sortie :

La *chaîne* `v2 :` sur la première ligne.

Sur les lignes suivantes, chaque élément de `v2` (un par ligne).

1.19.11.3 📌 Exemples

Entrée	Sortie	Observation
5 10 20 5 30 15	v2 : 20 20 30 30 30	À l'indice 1 : max(10, 20, 5) = 20

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

Figure 1.21: Simulateur EP01_11 : Filtre de Maximum Local 1D (Voisinage 1x3 avec Condition de Bord)

```
%%writefile EP01_11.py
# votre solution
```

```
Overwriting EP01_11.py
```

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

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

Chapitre 2

De la Capture au Pixel - Échantillonnage, Quantification et Connectivité

Exécuter Colab Ouvrir GitHub

Ce chapitre approfondit la compréhension de l'image numérique, passant de la nature physique de la capture à sa représentation mathématique discrète. Nous explorons comment la lumière devient donnée et comment l'organisation spatiale des pixels définit les relations de voisinage et de connectivité essentielles pour les algorithmes avancés de Vision par Ordinateur (VO).

2.1 Objectifs

À la fin de ce chapitre, vous serez capable de :

- Expliquer le modèle physique de formation de l'image basé sur l'éclairage et la réflectance.
- Différencier les mécanismes de la vision humaine et ceux des capteurs numériques.
- Comprendre les processus d'**échantillonnage** (discrétisation de l'espace) et de **quantification** (discrétisation de l'intensité).
- Décrire les relations topologiques entre pixels : voisinage, adjacence, connectivité et distances.
- Effectuer des transformations géométriques de base (translation, rotation, échelle) en préservant la qualité.
- Visualiser en pratique les effets de la variation de la résolution spatiale et de la profondeur de bits.

2.2 L'Œil et la Caméra - Éléments de la Perception Visuelle

La formation d'une image numérique commence par la capture de la lumière réfléchie par les objets. Comme illustré dans Figure 2.1, tant l'œil humain que les caméras numériques suivent des principes optiques similaires pour focaliser la lumière sur une surface sensible, bien qu'ils utilisent des mécanismes biologiques et électroniques distincts pour la transduction du signal.

2.2.1 Vision humaine

L'œil fonctionne comme un système optique complexe : la lumière traverse la cornée, l'humeur aqueuse, la pupille (contrôlée par l'iris) et le cristallin — qui ajuste la mise au point dynamiquement — jusqu'à atteindre la rétine. Sur la rétine se trouvent les photorécepteurs : les **cônes** (6 millions), concentrés dans la fovéa, sont responsables de la vision des couleurs et des détails, tandis que les **bâtonnets** (120 millions) assurent la vision en faible luminosité (vision scotopique), ne détectant que des intensités de gris. Le point aveugle est la région d'où part le nerf optique, dépourvue de récepteurs.

2.2.2 Capteurs numériques

Dans les caméras, les capteurs d'image jouent le rôle de la rétine. Les deux types les plus courants sont le **CCD** (*Charge-Coupled Device* - Dispositif à Transfert de Charge) et le **CMOS** (*Complementary Metal-*

Oxide-Semiconductor - Semi-conducteur à Oxyde Métallique Complémentaire). Le capteur est composé d'une matrice de photodiodes (pixels) qui accumulent une charge électrique proportionnelle à la lumière incidente.

Pour la reconstruction des couleurs, on utilise le **Filtre de Bayer**, une matrice de filtres colorés qui permet à chaque pixel de capturer uniquement une composante de couleur : rouge, vert ou bleu (RGGB - *Red, Green, Green, Blue*). Par la suite, un **ADC** (*Analog-to-Digital Converter* - Convertisseur Analogique-Numérique) quantifie cette charge en valeurs numériques, définies par une profondeur de bits (ex. : 8 bits, résultant en 256 niveaux d'intensité).

> **Curiosité**
>
> Bien que l'œil humain possède des millions de récepteurs, la résolution en haute définition est limitée à la fovéa (vision centrale), équivalente à environ 120×120 pixels. La perception d'une scène complète en haute résolution résulte d'un intense post-traitement effectué par le cerveau.

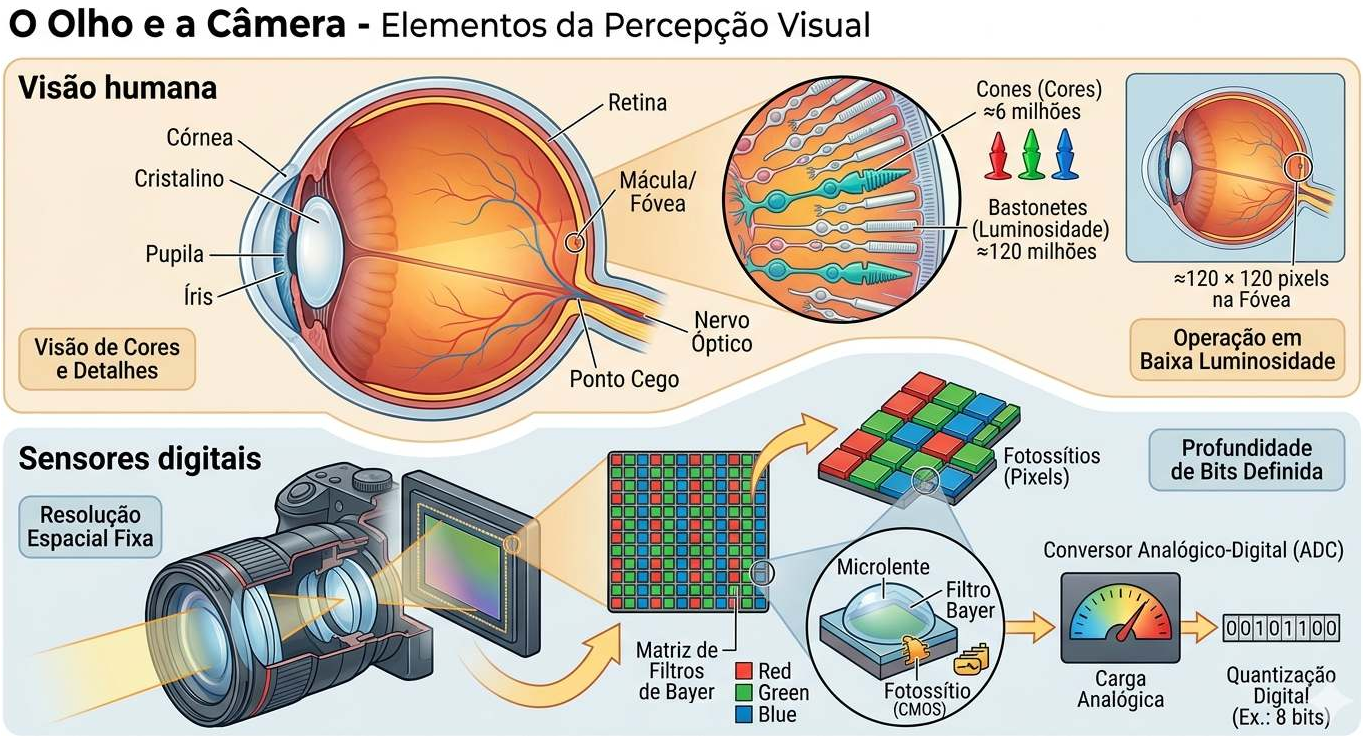


Figure 2.1: Comparaison didactique entre le système visuel biologique et le système électronique : en haut, l'anatomie de l'œil humain mettant en évidence la rétine et les photorécepteurs (cônes et bâtonnets) ; en bas, la structure d'un appareil photo numérique détaillant le capteur CMOS, la matrice de filtres de Bayer (RGGB) et le processus de quantification numérique réalisé par le convertisseur analogique-numérique (ADC).

2.3 Illusions d'optique : Les défis de la perception visuelle

Alors que les capteurs numériques capturent l'intensité de la lumière de manière linéaire et objective, le système visuel humain interprète la scène en se fondant sur le contexte, les expériences antérieures et les mécanismes biologiques de survie. Les illusions d'optique ne sont pas des « erreurs » de l'œil, mais des preuves de l'intense **post-traitement cérébral** réalisé dans le cortex visuel.

2.3.1 Ambiguïté et contexte

Le cerveau cherche constamment à donner un sens aux motifs ambigus. Dans l'exemple du **Vase de Rubin** (voir Figure 2.2), la perception alterne entre la figure (le vase) et le fond (deux visages), démontrant que nous ne pouvons pas traiter simultanément les deux interprétations. Quant à l'illusion de l'**Éléphant de**

Shepard, elle joue avec notre incapacité à concilier les lignes de contour qui suggèrent un volume dans des positions logiquement impossibles.

2.3.2 Luminosité et contraste local

De nombreuses illusions découlent de l'**inhibition latérale**, mécanisme par lequel les neurones voisins de la rétine se font concurrence pour accentuer les contours. Dans la **Grille scintillante**, des points sombres « fantômes » semblent apparaître aux intersections blanches en raison de ce traitement local du contraste.

L'illusion de l'**Ombre sur l'échiquier d'Adelson** est peut-être la plus frappante pour la VC : le carré « A » et le carré « B » ont exactement la même valeur de gris sur le capteur (ou dans le fichier numérique), mais le cerveau « corrige » la luminosité du carré « B » en comprenant qu'il se trouve sous une ombre portée, le percevant comme plus clair.

2.3.3 Géométrie et perspective

La perception de la profondeur peut être trompée par des constructions géométriques qui défient la logique tridimensionnelle à partir d'un angle de vue spécifique. L'**Escalier de Schröder** utilise l'ambiguïté de la perspective pour créer un objet qui semble monter ou descendre selon la façon dont on l'observe, mettant en évidence comment notre interprétation du « haut » et du « bas » dépend du point de fuite.

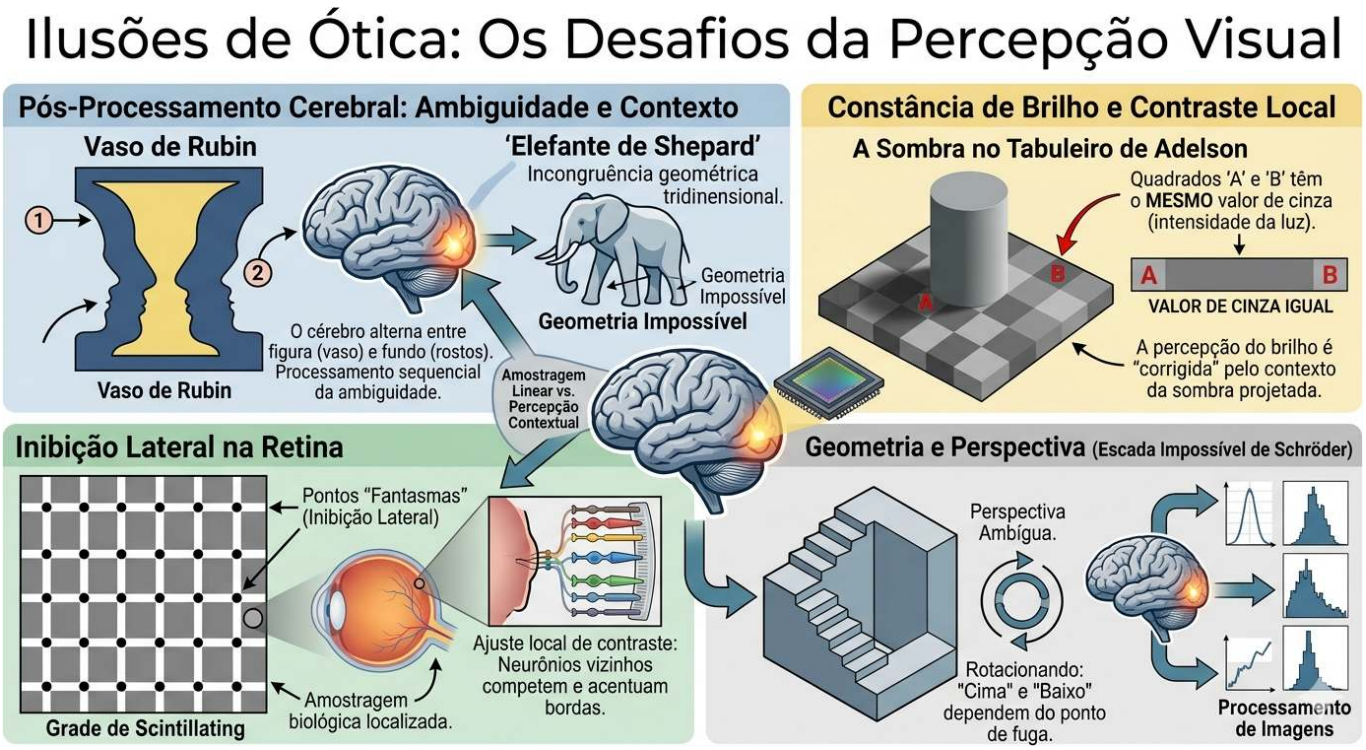


Figure 2.2: Recueil de défis perceptifs : (en haut à gauche) Vase de Rubin — ambiguïté figure-fond ; (en haut au centre) Éléphant de Shepard — incongruence géométrique ; (en haut à droite) Ombre d'Adelson — constance de la luminosité fondée sur le contexte ; (en bas à gauche) Grille de points — inhibition latérale ; (en bas à droite) Escalier de Schröder.

2.4 Le Modèle Mathématique de la Formation de l'Image

Une image peut être modélisée comme le produit de deux fonctions :

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

où :

- $i(x, y)$ est l'**éclairement** incident sur la scène (énergie lumineuse par unité de surface), déterminé par la source de lumière ;
- $r(x, y)$ est la **réflectance** de l'objet (fraction de la lumière réfléchie), déterminée par les propriétés optiques de la surface, avec $0 < r(x, y) < 1$.

En pratique, les deux composantes varient à des échelles spatiales distinctes : $i(x, y)$ tend à varier lentement dans l'espace, tandis que $r(x, y)$ peut présenter des variations brusques associées aux textures, aux contours et aux détails fins. Les techniques de traitement d'images (PDI) cherchent souvent à séparer ou à compenser ces composantes, comme dans les méthodes de correction d'éclairement non uniforme.

La Figure 2.3 illustre comment ce modèle se manifeste dans les images numériques en couleur, représentées par plusieurs canaux spectraux (RVB) et, éventuellement, par un canal supplémentaire de transparence (RVBA).

2.4.1 Représentation numérique et canaux spectraux

Dans les images numériques en couleur, la fonction $f(x, y)$ de la Équation 2.1 est représentée par de multiples canaux spectraux. Dans le standard **RGB**, chaque pixel stocke trois échantillons indépendants :

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

correspondant aux intensités des composantes rouge (*Red*), verte (*Green*) et bleue (*Blue*). Dans les images de type **RGBA**, on ajoute un quatrième canal :

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

où $A(x, y)$ représente le canal **alpha** (*Alpha*), chargé de coder l'opacité du pixel. Par convention, la valeur $A = 0$ indique un pixel entièrement transparent, tandis que $A = 255$ (ou 1) représente un pixel entièrement opaque.

Ainsi, une image numérique en couleur est structurée comme une matrice multidimensionnelle de dimensions :

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

où chaque position (x, y) stocke les échantillons associés aux propriétés optiques de cette coordonnée spatiale.

La conversion entre plages d'intensité est directe et donnée par :

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

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

Convention d'échelle entre bibliothèques : Différents écosystèmes de programmation adoptent des plages de valeurs et des ordres distincts pour représenter les canaux numériques, comme synthétisé dans la Table 2.1.

Tableau 2.1: Conventions d'échelle et ordre des canaux dans les principales bibliothèques de traitement d'images.

Bibliothèque	Type par défaut	Plage de valeurs	Ordre des canaux	Forme du tableau/tenseur
OpenCV	`uint8`	$[0, 255]$	**BGR**	$H \times W \times C$
Pillow	`uint8`	$[0, 255]$	RGB	$H \times W \times C$

Bibliothèque	Type par défaut	Plage de valeurs	Ordre des canaux	Forme du tableau/tenseur
NumPy	`uint8/float32`	$[0, 255]$ ou $[0, 1]$	—	$H \times W \times C$
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	$H \times W \times C$

Négliger ces divergences est une source fréquente d'erreurs dans les flux de travail de vision par ordinateur — comme, par exemple, injecter une image lue via OpenCV (format BGR, type `uint8`) directement dans un modèle profond de PyTorch qui présuppose le format RGB normalisé.

2.4.2 Préparation de l'environnement pratique

Le bloc suivant charge le module `morph.py` du dépôt et illustre, pour un pixel synthétique, les différentes conventions d'échelle et d'ordre des canaux adoptées par les principales bibliothèques de traitement d'images.

```
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
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0

 Conventions d'échelle par bibliothèque
 NumPy / Pillow / OpenCV (uint8) : [200 100  50] → [0, 255]
 scikit-image / PyTorch (float) :  [0.78431373 0.39215686 0.19607843] → [0.0, 1.0]
 OpenCV : attention - lecture en BGR : [ 50 100 200] (canaux inversés)
```

2.4.3 Implémentation de la Lecture d'Image dans `morph.py`

Le passage suivant présente le code source de la fonction `mm.read`, permettant de vérifier directement comment la bibliothèque `morph.py` implémente la lecture d'images.

```
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 et RGBA : Exemple pratique avec l'image Mandrill

L'exemple suivant charge l'image Mandrill au format RGB, construit artificiellement un canal alpha avec un dégradé horizontal et affiche les deux représentations, illustrant concrètement les structures matricielles $M \times N \times 3$ et $M \times N \times 4$.

```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"

#   Leitura 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

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

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

#   Visualização
```

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

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

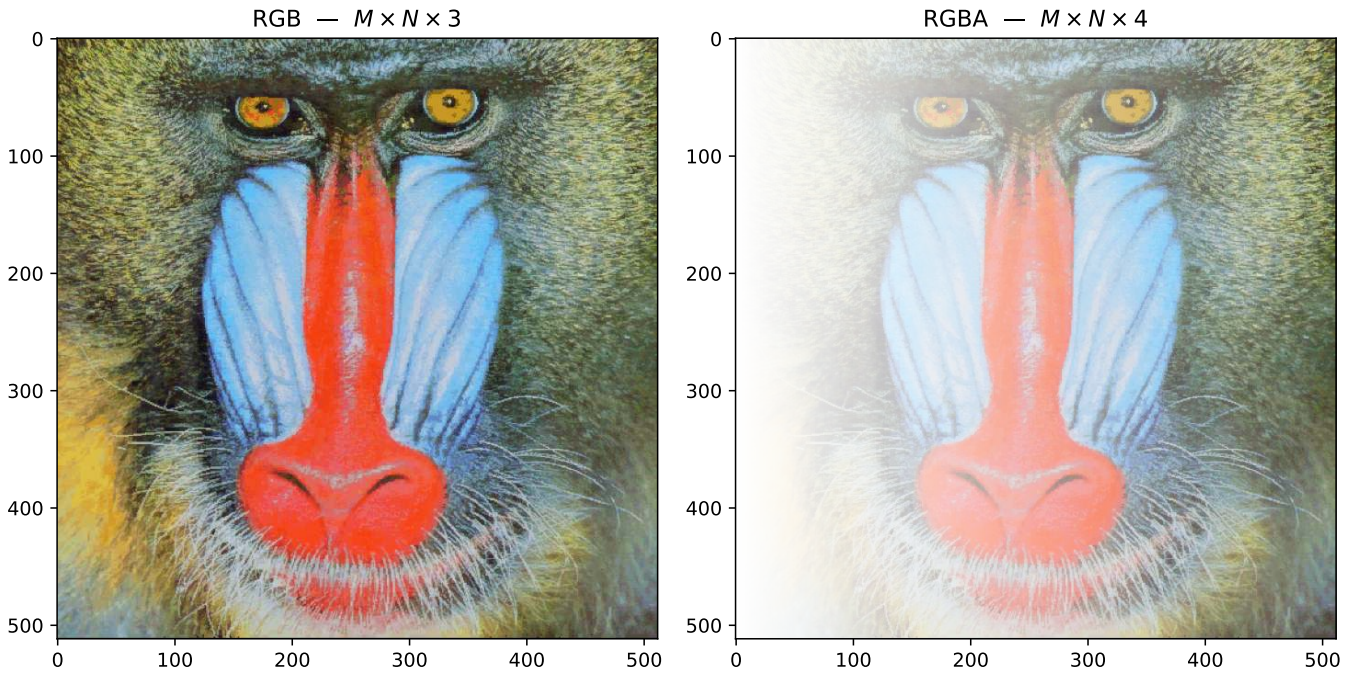


Figure 2.3: Imagem RGB (3 canais, $M \times N \times 3$) e versão RGBA (4 canais, $M \times N \times 4$) com transparência alfa gradual da esquerda para a direita. Imagem Mandrill — USC SIPI Image Database (domínio público).

2.5 Numérisation : Échantillonnage et Quantification

Pour transformer une scène continue en une image numérique, deux processus sont nécessaires : l'échantillonnage et la quantification.

2.5.1 Échantillonnage — Discrétisation de l'espace

L'échantillonnage consiste à mesurer la valeur de la fonction $f(x, y)$ en des points régulièrement espacés, formant une matrice de M lignes (hauteur) et N colonnes (largeur). Chaque élément de cette matrice est un **pixel**. La **résolution spatiale** est donnée par $M \times N$. Plus la résolution est élevée, plus les détails spatiaux sont préservés, mais plus le coût computationnel et de stockage augmente également.

2.5.2 Quantification — Discrétisation de l'intensité

La quantification associe à chaque pixel une valeur numérique discrète, généralement représentée par un entier de b bits. La **profondeur de bits** définit le nombre de niveaux d'intensité : 2^b. Les images en niveaux de gris utilisent couramment 8 bits (256 niveaux). Les images couleur utilisent trois canaux de 8 bits (24 bits au total).

Illustration : Si l'on utilise seulement 1 bit par pixel (noir et blanc), on perd tous les tons intermédiaires. Avec 2 bits (4 niveaux), on perçoit déjà des dégradés grossiers. Avec 8 bits, l'œil humain perçoit difficilement la discrétisation (vision continue).

> ⚠ Erreur de quantification
>
> L'**erreur de quantification** est la différence entre la valeur analogique réelle et la valeur discrète attribuée. Elle se manifeste sous forme de bruit de quantification, visible dans les régions à gradient doux lorsque l'on utilise peu de bits.

2.5.3 Effets de l'échantillonnage et de la quantification

Les expériences suivantes montrent comment la réduction de la résolution spatiale (sous-échantillonnage) et de la profondeur de bits dégrade la qualité visuelle. Utilisez le code pour explorer différents facteurs et niveaux de gris.

```
# Image d'exemple (barbu-rayé) - base des figures d'échantillonnage,
# quantification et transformations géométriques de ce chapitre.
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)  # recadrage pour voir les détails
print(f"Image originale : {img_color.shape}")
```

```
Image originale : (3067, 2047, 3)
```

```
import numpy as np

# Relecture comme objet PIL pour accéder aux métadonnées EXIF (Python uniquement).
img_pil   = mm.read(caminho, pil=True)   # retourne PIL Image avec EXIF
img_numpy = np.array(img_pil)
exif      = img_pil._getexif()           # ça fonctionne maintenant - img_pil est une PIL
Image
print(f"Type de l'image : {type(img_color)}")
```

```
Type de l'image : <class 'numpy.ndarray'>
```

L'expérience présentée dans Figure 2.4 illustre le compromis entre la **résolution spatiale** et le **coût de stockage** en mémoire. Le code utilise la technique de sous-échantillonnage par tranchage (*slicing*) pour réduire la matrice originale de pixels selon un facteur f, ce qui entraîne une économie de mémoire — par exemple, un facteur $f = 8$ réduit la taille des données de 64 fois (8^2). À des fins de comparaison visuelle, les images réduites sont restaurées à leurs dimensions originales (512×512) via une interpolation par **plus proche voisin** (`nearest`). Ce processus ne récupère pas l'information perdue, mais rend évident l'effet d'***aliasing*** et la structure en blocs (pixelisation) générée par la faible densité de données de la matrice échantillonnée.

```python
def subsample_simple(image, f):
    # Sous-échantillonnage par découpage (slicing)
    reduced = mm.subsample(image, f)

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

    # Restaure la taille pour la visualisation (H, W d'origine)
    res = mm.resize(reduced, (image.shape[1], image.shape[0]), method='nearest')
    return res, label

factors = [1, 4, 8, 12]

# Génère les résultats et les sépare en listes pour mm.show
results = [subsample_simple(img_gray, f) for f in factors]
imgs_list = [r[0] for r in results]
titles_list = [r[1] for r in results]

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

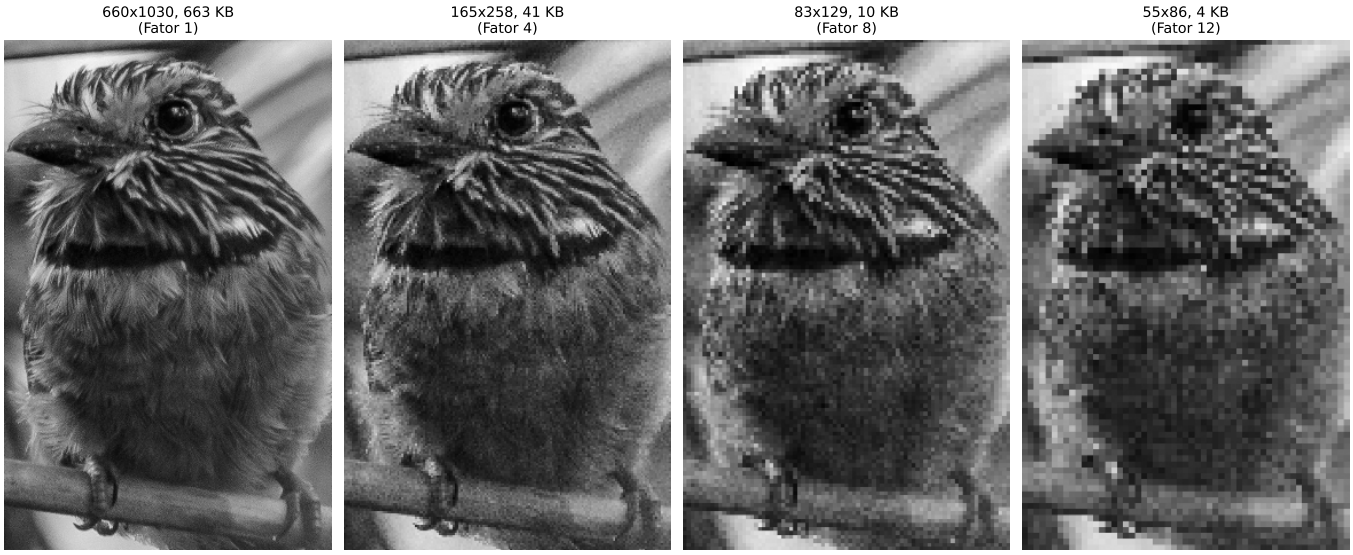


Figure 2.4: Effet du sous-échantillonnage. Les titres affichent les dimensions (W x H) et la taille de la matrice en mémoire (Ko).

L'expérience dans Figure 2.5 se concentre sur la **quantification d'intensité**, le processus de discrétisation de l'amplitude de la fonction $f(x, y)$. Alors que le sous-échantillonnage affecte la grille spatiale, la réduction de la profondeur de bits limite la quantité de niveaux de gris disponibles pour représenter la luminosité.

En réduisant la profondeur de 8 bits (256 niveaux) à des valeurs inférieures, l'effet de **postérisation** apparaît, où les dégradés lisses d'une scène sont remplacés par des transitions abruptes. À la limite de 1 bit, l'image devient strictement binaire, ne préservant que la silhouette et perdant les détails de texture et de volume.

```python
def quantize_simple(image, bits):
    """Réduit la profondeur de bits et calcule les métadonnées de mémoire."""
    levels = 2 ** bits
    # Normalisation et quantification uniforme
    quantized = (np.floor(image / 256 * levels) / levels * 255).astype(np.uint8)

    # Calcul de la mémoire en Ko
    mem_kb = quantized.nbytes / 1024
```

```
    label = f"{bits} bits ({levels} níveis)\n{int(mem_kb)} KB"

    return quantized, label

# Liste de bits à tester (8 est la valeur par défaut, 1 est le binaire)
bits_test = [8, 4, 2, 1]

# Génère les résultats et les sépare en listes pour mm.show
results_q = [quantize_simple(img_gray, b) for b in bits_test]
imgs_q = [r[0] for r in results_q]
titles_q = [r[1] for r in results_q]

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

Figure 2.5: Effet de la réduction de la profondeur de bits. Les titres affichent la quantité de bits, les niveaux et la taille en mémoire (Ko).

2.5.4 Analyse technique

- **Domaine vs. Codomaine** : Notez que la résolution spatiale (dimensions de la matrice) reste constante à 512x512 ; seul le codomaine de la fonction image change.
- **Constance de la mémoire** : Observez dans les titres que la taille en Ko ne diminue pas. Cela s'explique par le fait que NumPy stocke chaque pixel quantifié dans un conteneur de 8 bits (`uint8`), indépendamment du fait que la valeur réelle soit seulement 0 ou 1.
- **Perception** : La dégradation visuelle devient critique en dessous de 4 bits, où l'œil humain commence à percevoir les « frontières » artificielles créées par le manque de tons intermédiaires.

La limitation des types de données plus petits qu'un octet dans l'écosystème Python/NumPy est due à l'architecture matérielle, qui adresse la mémoire en blocs de 8 bits (octets). Afin de maintenir la compatibilité avec OpenCV et de garantir l'efficacité, même les éléments binaires sont mappés vers des conteneurs de 1 octet (`uint8` ou `bool8`).

Bien que des langages comme l'ANSI C permettent la compression de 8 pixels par octet (*bit-packing*), cette approche exige une décompression constante pour les calculs et impose une complexité élevée dans la manipulation des pointeurs. Selon la Table 2.2, l'utilisation de `uint8` est privilégiée pour la facilité d'accès aux voisins et la polyvalence dans les transformations géométriques. De plus, les méthodes natives de NumPy et OpenCV exécutent le traitement en interne à bas niveau (C/C++), ce qui rend les opérations vectorisées plus rapides que les implémentations manuelles avec des boucles imbriquées en Python.

Tableau 2.2: Comparatif entre les stratégies de compression et l'efficacité du traitement.

Caractéristique	Python (NumPy/OpenCV)	ANSI C (Bit-packing)
Plus petite unité	1 octet (8 bits)	1 bit
Mémoire (binaire)	256 Ko (pour 512x512)	32 Ko (pour 512x512)
Vitesse	Élevée (vectorisation en C)	Variable (lente en cas de décalage de bits)
Complexité	Faible : méthodes prêtes à l'emploi	Élevée : pointeurs et masques

2.6 Relations entre Pixels - Topologie de l'Image

Les pixels ne sont pas des éléments isolés ; leurs positions relatives définissent des concepts importants pour le traitement.

2.6.1 Voisinage

Étant donné un pixel de coordonnées (x, y), on définit deux types principaux de voisinage (pour les images en grille rectangulaire) :

- **Voisinage-4** (von Neumann) : inclut les pixels aux positions $(x-1, y)$, $(x+1, y)$, $(x, y-1)$, $(x, y+1)$.
- **Voisinage-8** (Moore) : inclut les huit pixels adjacents (ajoute les quatre diagonales).

Le choix du voisinage influence des opérations telles que la détection de contours, le calcul des gradients et la connectivité.

```
# # Création d'une matrice 3x3 pour l'exemple topologique
# viz = np.zeros((3, 3), dtype='uint8')

# # Définition du voisinage-4 (N4) avec une valeur différente pour la mise en évidence
# viz[0, 1] = viz[2, 1] = viz[1, 0] = viz[1, 2] = viz[1, 1] = 255

# ou simplement (testez avec des nombres comme arguments) :
viz = mm.secross()

# Affichage de la matrice pour l'analyse des coordonnées
mm.drawImgPlt(viz, scale=40)
```

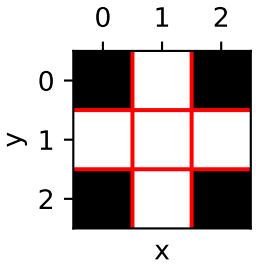


Figure 2.6: Illustration des voisinages 4 dans une matrice 3x3. Au centre (1,1), le pixel d'intérêt.

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

2.6.2 Adjacence, connectivité et chemins

Deux pixels sont **adjacents** s'ils sont en contact selon un voisinage défini **et** satisfont un critère de valeur (ex. : même niveau d'intensité). Une **connectivité** définit une relation d'équivalence entre les pixels qui forment une région connexe. Un **chemin** est une séquence de pixels adjacents.

La **connectivité-4** (N4) et la **connectivité-8** (N8) peuvent produire des résultats différents dans la segmentation et le calcul des composantes connexes (étiquetage). Par exemple, un motif en damier peut être complètement déconnecté en N4, mais totalement connecté en N8.

2.6.3 Distances entre pixels

Les métriques de distance sont fondamentales pour quantifier la proximité physique et la connectivité entre les éléments qui composent la grille numérique. Comme démontré dans la Table 2.3, le choix de la métrique définit le coût de déplacement entre pixels et modifie le comportement des algorithmes de segmentation et d'analyse morphologique.

Pour mesurer la distance entre deux pixels $p(x_1, y_1)$ et $q(x_2, y_2)$, on utilise différentes fonctions métriques qui imposent des contraintes de mouvement distinctes sur la *grille* :

Tableau 2.3: Comparatif des métriques de distance appliquées à la maille de pixels.

Métrique	Définition	Interprétation
Euclidienne	$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$	Distance **exacte** en ligne droite (continue)
Manhattan (*City block*)	$\lvert x_1 - x_2\rvert + \lvert y_1 - y_2\rvert$	Mouvements horizontaux + verticaux
Chebyshev (*Échiquier*)	$\max(\lvert x_1 - x_2\rvert, \lvert y_1 - y_2\rvert)$	Plus grand déplacement entre les axes

Ces distances sont appliquées dans divers contextes de TDI, notamment les algorithmes d'interpolation géométrique, les transformations de distance, la croissance de régions et l'analyse de formes.

> **Exemple pratique**
>
> Considérant deux pixels avec des déplacements relatifs $\Delta x = 3$ et $\Delta y = 4$:
> - **Euclidienne** : $\sqrt{3^2 + 4^2} = 5$ (hypoténuse du triangle rectangle).
> - ***Manhattan*** : $3 + 4 = 7$ (somme des cathètes).
> - ***Chebyshev*** : $\max(3, 4) = 4$ (prédominance du plus grand déplacement).

2.7 Stockage d'images

Le choix du format de fichier est une étape décisive dans le flux de traitement, car il détermine la manière dont les données d'échantillonnage et de quantification seront préservées ou écartées. Comme présenté dans la Table 2.4, chaque extension équilibre de façon distincte la fidélité des données et l'efficacité de stockage.

Tableau 2.4: Principaux formats de stockage d'images numériques et leurs applications en TNI.

Format	Caractéristiques	Usage typique
PGM	Format simple de carte de niveaux de gris (texte ou binaire).	Recherche académique et outils Unix.
BMP	Non compressé (ou compression simple).	Windows, applications héritées.
PNG	Compression **sans perte** (*lossless*).	Web, images avec transparence.

Format	Caractéristiques	Usage typique
JPEG	Compression **avec perte** (*lossy*), idéale pour les photographies.	Photos, appareils photo numériques.
TIFF	Prend en charge plusieurs couches et une compression variée.	PAO, archivage.
RAW	Données brutes du capteur, sans traitement.	Photographie professionnelle.
DICOM	Norme médicale avec métadonnées cliniques intégrées (patient, équipement, protocole).	Radiologie, tomodensitométrie, imagerie par résonance magnétique.

Les métadonnées d'une image incluent des paramètres tels que la largeur, la hauteur, la profondeur de bits et le codage des couleurs. Dans les formats scientifiques, les informations d'étalonnage et les détails de capture sont également préservés. Lors de l'utilisation de la fonction `mm.read()`, la bibliothèque `morph` préserve automatiquement ces données afin de respecter les propriétés originales de l'image.

Dans les contextes scientifiques et hospitaliers, la norme **DICOM** (*Digital Imaging and Communications in Medicine*) est privilégiée pour garantir qu'aucune perte de précision diagnostique ne se produise. Des référentiels publics tels que **The Cancer Imaging Archive (TCIA)**, le **Alzheimer's Disease Neuroimaging Initiative (ADNI)** et **PhysioNet** mettent à disposition de vastes ensembles de données dans ce format, incluant des métadonnées cliniques anonymisées qui sont essentielles pour la recherche scientifique.

2.7.1 Exemple : Extraction de métadonnées et localisation GPS

Contrairement à la matrice de pixels pure obtenue par la lecture conventionnelle — comme dans l'image de l'oiseau présentée au début de ce chapitre —, l'utilisation de l'argument `pil=True` dans la méthode `mm.read()` modifie la nature de l'objet retourné (voir Figure 2.7). Alors que le comportement par défaut (`pil=False`) retourne un `numpy.ndarray` RGB, la lecture avec `pil=True` retourne un objet spécialisé de la bibliothèque Pillow, capable d'interpréter l'en-tête **EXIF**.

L'en-tête **EXIF** (*Exchangeable Image File Format*) fonctionne comme un référentiel technique de la capture, permettant à l'objet Pillow d'interpréter une vaste gamme d'informations qui vont bien au-delà des coordonnées GPS. En utilisant `pil=True`, le système accède à l'« ADN » de l'image, incluant les métadonnées matérielles (marque et modèle de l'appareil photo), les réglages optiques (ouverture, distance focale et temps d'exposition) et les paramètres d'éclairage (utilisation du flash et balance des blancs).

Cette distinction est importante en traitement d'images numériques, car elle transforme la matrice d'échantillonnage en un ensemble de données contextualisées, où les caractéristiques physiques du capteur et de l'objectif peuvent être utilisées pour normaliser les luminosités ou corriger les distorsions géométriques.

```
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. Lecture - préserve l'objet PIL avec EXIF
if not os.path.exists(caminho):
    os.makedirs("imagens", exist_ok=True)
    img_obj = mm.read(url, pil=True)
```

```python
    mm.write(img_obj, caminho)      # sauvegarde en préservant EXIF
else:
    img_obj = mm.read(caminho, pil=True)

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

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

# 3. Diagnostic des types, dimensions et accès aux pixels
print(f"\nType PIL  : {type(img_obj)}   | Dimensions (x, y) : {img_obj.size}")
print(f"Pillow (0,0) : {img_obj.getpixel((0, 0))}")
print(f"Type NumPy : {type(img_numpy)} | Dimensions [y, x, c] : {img_numpy.shape}")
print(f"NumPy [0,0] : {img_numpy[0, 0]}")

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

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

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

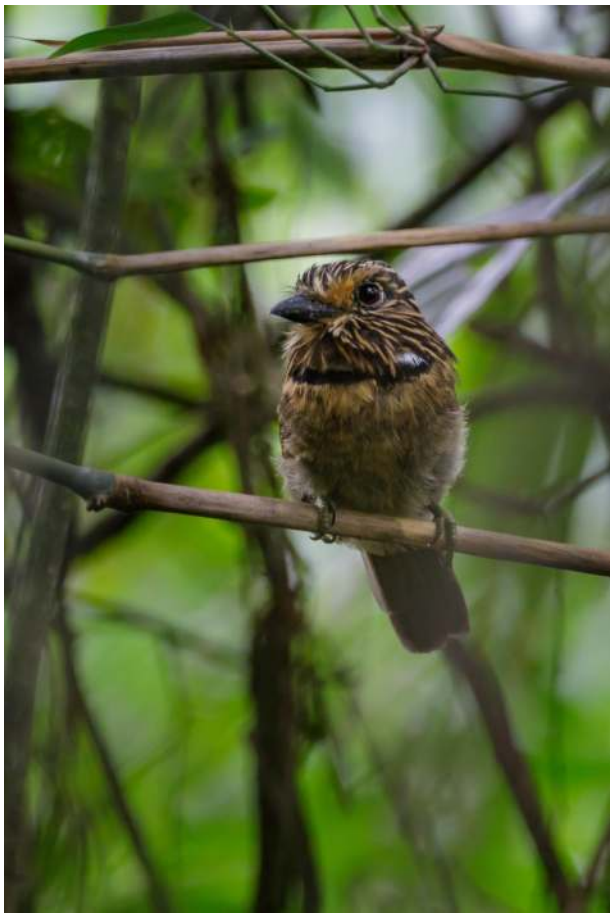


Figure 2.7: Zone de Protection Environnementale Quilombos do Médio Ribeira - Barbudo-rajado (Malacoptila striata). Crédit : Thomas Fuhrmann (CC BY-SA 4.0).

> **Note pédagogique : La différence subtile des dimensions**
>
> Remarquez que la représentation des dimensions change selon la structure de données utilisée :
> - **Dans Pillow (`.size`)** : Renvoie `(Largeur, Hauteur)` — dans l'exemple : `(2047, 3067)`. C'est une vision orientée vers le fichier image.
> - **Dans NumPy (`.shape`)** : Suit la convention mathématique des matrices : `(Lignes/Hauteur, Colonnes/Largeur, Canaux)` — dans l'exemple : `(3067, 2047, 3)`.
>
> Cette distinction est fondamentale pour éviter les erreurs d'indexation lors de l'implémentation de filtres manuels. Alors que l'objet Pillow porte le « **où** » et le « **quand** » (contexte), le tableau NumPy porte le « **combien** » de lumière (intensité) existant à chaque point de l'image.

2.7.2 Pourquoi cette séparation est-elle importante ?

Lors du chargement d'une image par le chemin conventionnel (`pil=False`), le résultat est un `numpy.ndarray`, qui contient strictement les valeurs numériques issues de la **quantification** et de l'**échantillonnage**. Cependant, en utilisant `pil=True`, `mm.read()` renvoie un objet de la classe `PIL.JpegImagePlugin.JpegImageFile`.

Cette classe maintient le fichier « ouvert » pour permettre l'accès au contexte de la capture avant que les données ne soient converties en matrice brute. Notez que le pixel `(0, 0)` est identique dans les deux représentations — `(58, 96, 0)` dans Pillow et `[58, 96, 0]` dans NumPy —, confirmant que les deux décrivent les mêmes données, avec des interfaces simplement différentes. Cette séparation est vitale : les pixels servent aux algorithmes ; les métadonnées servent au géoréférencement, au catalogage scientifique et aux corrections basées sur le matériel d'acquisition.

Pour inspecter toutes les métadonnées EXIF d'un objet 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 Transformations géométriques de base

Les transformations géométriques modifient la position des pixels, en conservant les valeurs d'intensité. Elles sont fondamentales pour l'alignement, la correction des distorsions et l'augmentation des données (*data augmentation*) en apprentissage automatique.

Une **transformation affine** est toute application qui préserve la colinéarité (des points sur une droite restent sur une droite) ainsi que les rapports de distances entre points colinéaires. En 2D, toute transformation affine peut être exprimée en **coordonnées homogènes** par une matrice 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 sous-matrice 2×2 supérieure gauche $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ contrôle la rotation, l'échelle et le cisaillement ; le vecteur $(t_x, t_y)^\top$ contrôle la translation. Les transformations géométriques les plus utilisées en TNI — translation, rotation et échelle — sont des cas particuliers de $\mathbf{T}$, et peuvent être **composées** par multiplication matricielle, dans l'ordre $\mathbf{T} = \mathbf{T}_n \cdots \mathbf{T}_2 \mathbf{T}_1$.

> **ℹ Transformation inverse et interpolation**
>
> Dans l'implémentation pratique (`cv2.warpAffine`), on applique la **transformation inverse** : pour chaque pixel (x', y') de l'image destination, on calcule la position d'origine $(x, y) = \mathbf{T}^{-1}(x', y')$ et on interpole la valeur. Cela évite les trous dans l'image résultante, causés par le mappage direct de pixels entiers vers des positions non entières.

2.8.1 Translation

La translation est la transformation affine la plus simple : elle déplace tous les pixels selon un vecteur (t_x, t_y). En coordonnées homogènes, elle s'exprime par la matrice :

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

La troisième ligne de la matrice garantit que l'opération reste dans l'espace affine, permettant de combiner translation, rotation et mise à l'échelle par simple multiplication de matrices. En pratique, `cv2.warpAffine` n'utilise que les deux premières lignes (matrice 2×3), car la troisième est toujours $[0, 0, 1]$.

Les pixels déplacés au-delà de la zone d'origine sont ignorés ; les zones découvertes sont remplies avec 0 (noir). Voir un exemple dans la Figure 2.8..

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

Figure 2.8: Exemple de translation de l'image de l'oiseau avec des déplacements (50,50) et (100,50).

2.8.2 Rotation

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

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

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

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

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

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

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

Figure 2.9: Exemple de rotation de l'image de l'oiseau à 30° et 45° en utilisant l'interpolation bilinéaire.

2.8.3 Échelle (Redimensionnement)

Le redimensionnement par facteurs (s_x, s_y) est une transformation affine de matrice :

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

Lorsque $s > 1$ (agrandissement), les pixels de l'image cible correspondent à des positions non entières dans l'image source — ce qui exige une interpolation pour estimer la valeur. Lorsque $s < 1$ (réduction), plusieurs pixels source contribuent à un seul pixel cible — ce qui exige une **décimation**. Les trois mêmes méthodes décrites pour la rotation sont disponibles dans `mm.resize`, à la différence près qu'ici l'impact visuel est plus perceptible : en agrandissement, `nearest` produit un effet de blocs (*pixelation*), tandis que `bicubic` préserve mieux la netteté des contours, conformément à Table 2.5:

Tableau 2.5: Méthodes d'interpolation disponibles dans mm.resize et leurs nombres respectifs de voisins utilisés dans le calcul.

Méthode	Voisins utilisés	Caractéristique
`'nearest'`	1	Rapide ; produit un effet de blocs (*pixelation*)
`'bilinear'`	4	Bon compromis qualité/coût ; contours lisses
`'bicubic'`	16	Netteté accrue ; préférée dans les logiciels professionnels

Le paramètre `size_or_factor` accepte aussi bien un scalaire (facteur uniforme, e.g. `0.5` pour réduire de moitié) qu'un tuple `(largeur, hauteur)` pour des dimensions absolues.

```
# 1. Recadrage de la région du bec
y, x, offset = 210, 40, 40
crop = img_gray[y-offset:y+offset, x-offset:x+offset]
print(f"Image : {img_gray.shape} | Recadrage : {crop.shape}")

# 2. Agrandir 4× avec 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. Affichage comparatif
mm.show(
    [crop, crop_nearest, crop_bilinear],
    titles=["Original (recadrage)", "Plus proche voisin (4×)", "Bilinéaire (4×)"],
    cols=3, figsize=(16, 12), dpi=200
)
```

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

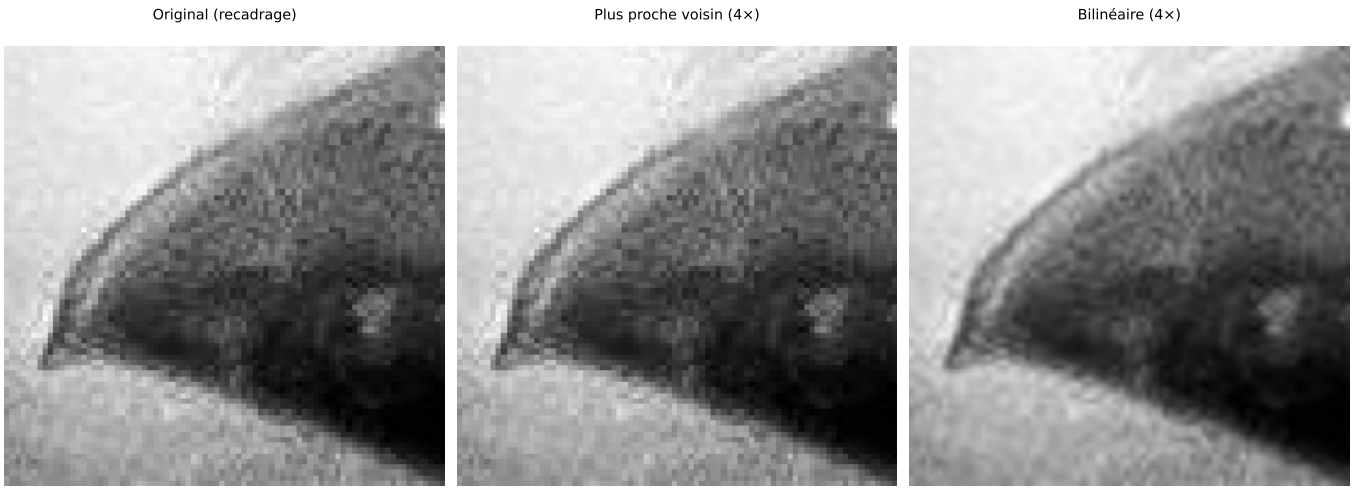


Figure 2.10: Comparaison de l'interpolation avec zoom sur le détail de l'œil (recadrage 60×60, agrandi 4×). Notez l'effet de blocs dans le plus proche voisin vs. l'adoucissement en bilinéaire.

2.8.4 Cisaillement (*Shear*)

Le cisaillement est une transformation affine qui déforme l'image en déplaçant chaque pixel proportionnellement à sa position sur un axe. La matrice générale combine un cisaillement horizontal (sh_x) et un cisaillement vertical (sh_y) :

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

Pour $sh_x \neq 0$ et $sh_y = 0$, chaque ligne est déplacée horizontalement proportionnellement à sa position verticale — produisant l'effet d'« inclinaison » caractéristique. Voir l'exemple dans la Figure 2.11.

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

mm.show(
    [img_gray, img_shx, img_shy, img_shc],
    titles=["Original", "Horiz. (shx=0.3)", "Vert. (shy=0.3)", "Combiné (0.2, 0.2)"],
```

```
    cols=4, figsize=(16, 12)
)
```

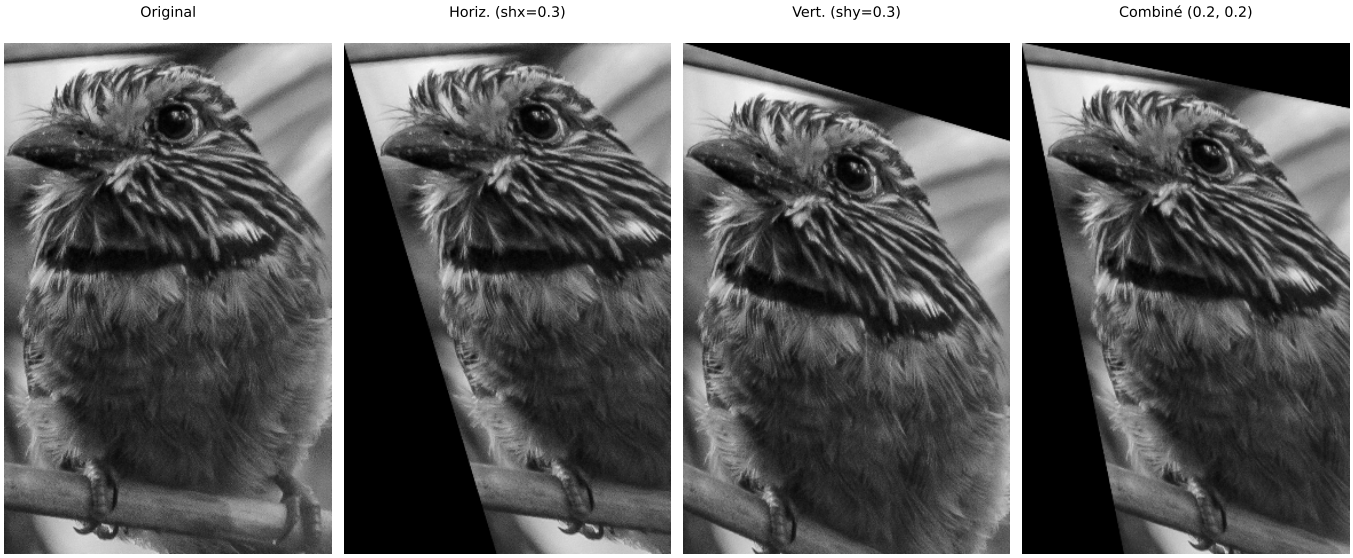


Figure 2.11: Exemple de cisaillement de l'image de l'oiseau : horizontal (shx=0.3), vertical (shy=0.3) et combiné (shx=0.2, shy=0.2).

2.9 Résumé

Dans ce chapitre, les principes fondamentaux de la numérisation et de la topologie des images ont été présentés :

- **Formation de l'image :** $f(x, y) = i(x, y) \cdot r(x, y)$.
- **Échantillonnage :** discrétisation de l'espace → résolution spatiale $M \times N$.
- **Quantification :** discrétisation de l'intensité → profondeur de bits b.
- **Relations topologiques :** voisinage-4, voisinage-8, connectivité, distances (euclidienne, Manhattan, Chebyshev).
- **Transformations géométriques :** translation, rotation, mise à l'échelle (avec interpolation bilinéaire ou plus proche voisin).
- **Formats de fichiers :** BMP, PNG, JPEG, TIFF, RAW ; chacun présentant différents compromis entre qualité et taille.

Le chapitre 3 abordera les **opérations spatiales** telles que la convolution, le filtrage et la morphologie mathématique (érosion, dilatation).

2.10 🤖 Utilisation de Gemini Notebook comme tuteur complémentaire

Dans cette édition, nous encourageons l'utilisation de **Gemini Notebook** comme outil d'apprentissage complémentaire. Cet outil d'IA utilise exclusivement les documents fournis par l'auteur comme base de connaissances, garantissant des réponses cohérentes avec le contenu du livre.

Pour chaque chapitre, nous avons préparé un projet spécifique sur la plateforme. Pour une expérience d'étude enrichie, utilisez l'accès ci-dessous :

🎓 Étudiez avec le tuteur intelligent

Pour interagir avec le contenu de ce chapitre, accédez au lien suivant. L'environnement contient des supports pédagogiques dans différents formats, générés à partir du **PDF** du chapitre. Sur la

plateforme, explorez particulièrement les options **Guide d'étude** et **Conversation** pour approfondir votre compréhension.
🚀 ACCÉDER À GEMINI NOTEBOOK : CHAPITRE 02

🌐 **Langue et langage de programmation**

Le projet de ce chapitre dans Gemini Notebook a été construit uniquement avec le texte en **portugais** et les exemples de code en **Python**. Si vous étudiez avec l'édition en anglais ou en français, ou si vous suivez le parcours en C++, les réponses du tuteur peuvent ne pas correspondre exactement à la version que vous lisez.

⚠️ **Avertissement concernant le contenu généré par l'IA**

L'IA est une alliée puissante dans les études, mais le contenu généré peut contenir des **erreurs ou des imprécisions**. Consultez toujours **des livres, des articles scientifiques et d'autres sources académiques fiables** pour valider les informations. Chaque fois que possible, exécutez les exemples pratiques fournis dans ce chapitre pour vérifier les résultats.

2.11 Liste d'exercices

1. **(15 %)** Expliquez, avec vos propres mots, la différence entre **échantillonnage** et **quantification**. Donnez un exemple concret de chacun dans le contexte d'une image numérique.
2. **(15 %)** Considérez une image avec une résolution spatiale de 1024×768 pixels et une profondeur de 24 bits (8 bits par canal RVB). Calculez la taille totale non compressée de l'image en octets et en mégaoctets.
3. **(20 %)** À l'aide du code du laboratoire, modifiez le facteur de sous-échantillonnage à 3 et à 6. Décrivez visuellement ce qui se produit au niveau des bords des objets. Qu'est-ce que l'effet de *crénelage* (*aliasing*) ?
4. **(20 %)** Pour l'image en niveaux de gris, appliquez une quantification avec 3 bits (8 niveaux) et 5 bits (32 niveaux). Comparez les résultats et expliquez pourquoi 5 bits peuvent déjà être considérés comme suffisants pour de nombreuses applications.
5. **(15 %)** Étant donné deux pixels $A = (10, 20)$ et $B = (15, 25)$, calculez les distances euclidienne, de Manhattan et de Tchebychev entre eux.
6. **(15 %)** À l'aide de la fonction `mm.rotate`, faites pivoter l'image de l'oiseau selon des angles de 90°, 180° et 270° avec une interpolation bilinéaire. Comparez avec la rotation utilisant `method='nearest'`. Dans quelles situations l'interpolation du plus proche voisin est-elle encore utile ?

Références du chapitre

La base théorique de ce chapitre s'appuie sur les ouvrages suivants :

- Gonzalez (2018) pour les concepts d'échantillonnage, de quantification et les relations entre pixels.
- Szeliski (2022) pour les transformations géométriques et la connectivité.
- Bradski (2008) pour l'implémentation pratique avec OpenCV et `morph.py`.

Exécuter Colab | Ouvrir GitHub

2.12 💻 Partie Pratique avec Exercices de Programmation

🎯 Objectif de ce cahier

Le cahier permet de développer, valider, organiser et tester des solutions d'**Exercices de Programmation (EPs)** dans des environnements interactifs, comme Colab, avec les mêmes cas de test que Moodle, en y copiant uniquement au moment d'enregistrer la note officielle.

Téléchargement

Téléchargez `morph.py` et `testsuite.py` en exécutant la cellule ci-dessous :

```
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
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0 | TestSuite : 1.1.2
```

Exécution des tests

Pour évaluer les tests, exécutez `TestSuite("EP04_01.extensão").run()` dans une nouvelle cellule, en remplaçant l'extension par celle du langage utilisé (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). Le système télécharge les cas de test depuis GitHub, exécute le programme et calcule automatiquement la note.

Pour tester directement du code Python, sans enregistrer de fichier, utilisez `run_code(codigo)` en passant le code comme *chaîne de caractères* dans une variable `codigo` :

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

2.12.1 EP02_01 ⊙ Réglage de la luminosité et du contraste

Dans cette activité, l'objectif est d'implémenter un opérateur ponctuel pour la **transformation linéaire d'intensité**, en appliquant le réglage dynamique de la luminosité et du contraste sur une image numérique.

2.12.1.1 📋 Directives d'implémentation

L'algorithme doit suivre le flux d'exécution ci-dessous :

1. **Dimensions :** Lire les entiers L (lignes) et C (colonnes) de la matrice.
2. **Paramètres :** Lire la valeur réelle α (facteur de contraste) et l'entier β (facteur de luminosité).
3. **Données :** Lire les valeurs entières de la matrice d'origine.
4. **Mappage :** Pour chaque pixel p, calculer la nouvelle valeur p' à l'aide de l'équation :

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

5. **Sortie :** Afficher la matrice résultante avec les dimensions $L \times C$.

2.12.1.2 📌 Contraintes informatiques

- **Arrondi (*Round*) :** On applique l'arrondi mathématique à l'entier le plus proche avant la conversion de type.
- **Saturation (*Clipping*) :** Les valeurs doivent être confinées à l'intervalle $[0, 255]$ pour préserver le standard 8 bits :

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

- **Simulation :** L'effet des paramètres α et β sur la correction histogrammique peut être observé dans la Figure 2.12.

2.12.1.3 🧠 Fondements théoriques

Les modifications altèrent l'histogramme de l'image pour ajuster le profil d'éclairage et la distinction tonale.

Paramètre	Fonction	Impact visuel
α (***Alpha***)	Scalaire	Module le **Contraste**. Si $\alpha > 1$, il étend l'histogramme ; si $0 \leq \alpha < 1$, il le compresse.
β (***Beta***)	Additif	Module la **Luminosité**. S'il est positif, il translate l'histogramme vers la droite ; s'il est négatif, vers la gauche.
clip	Limiteur	Restreint la plage dynamique, empêchant les erreurs de *sous-dépassement* et de *dépassement*.

2.12.1.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Valeurs de `alpha` (α) et `beta` (β).
- Lignes suivantes : Éléments numériques de la matrice d'origine.

Sortie :

- Matrice transformée structurée en L lignes et C colonnes.

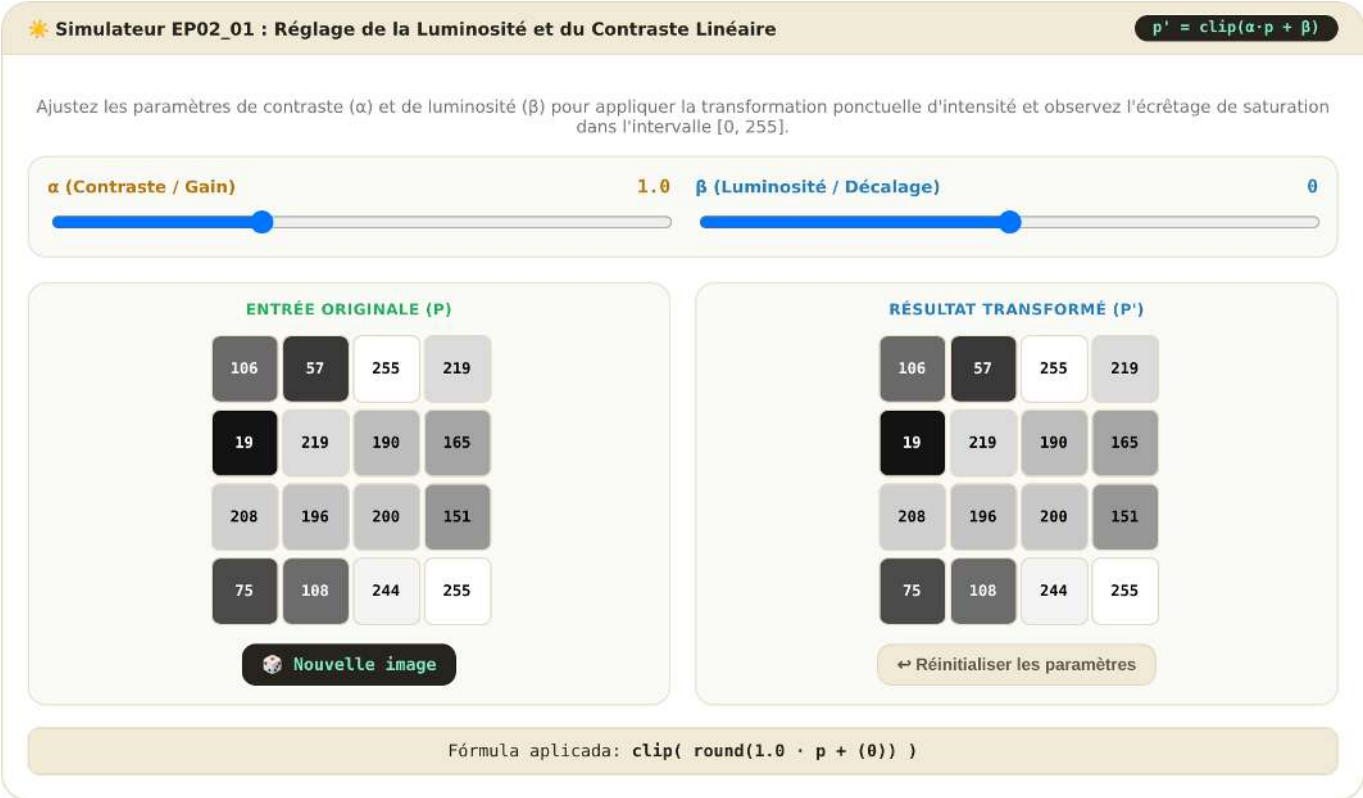
2.12.1.5 📌 Exemples

Le tableau suivant présente un exemple pratique du comportement attendu de l'algorithme, mettant en évidence l'action des opérateurs d'arrondi et de saturation.

Entrée	Sortie	Observation
1 4 1.5 -30 0 100 180 255	0 120 240 255	Notez l'effet de saturation sur le dernier pixel

Exécution des Tests

Pour évaluer les tests, exécutez `TestSuite("EP02_01.extension").run()` dans une nouvelle cellule, en remplaçant l'extension par celle du langage utilisé (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). Le système télécharge les cas de test depuis GitHub, exécute le programme et calcule la note automatiquement.



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

Figure 2.12: Simulateur EP02_01 : Ajustement de la Luminosité et du Contraste Linéaire (p' = p +)

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

```
Overwriting EP02_01.py
```

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

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

2.12.2 EP02_02 🔬 Sous-échantillonnage spatial

Dans cette activité, vous devez implémenter la réduction de la résolution spatiale d'une image par le processus de sous-échantillonnage.

- Lisez deux entiers **L** et **C**, représentant les dimensions de la matrice d'origine.
- Lisez une valeur entière f ($f \geq 1$), qui représente le facteur d'échantillonnage.
- Lisez les valeurs entières de la matrice d'origine.
- La nouvelle image doit être construite en sélectionnant le pixel de la position $(f \cdot i, f \cdot j)$ de l'image d'origine.
- Imprimez la matrice résultante avec les nouvelles dimensions.
- Voir dans Figure 2.13 une simulation de cet EP.

📌 **Important** :

- **Dimensions finales :** L'image échantillonnée aura pour dimensions $\lceil L/f \rceil \times \lceil C/f \rceil$. Dans le contexte de la programmation, cela équivaut à la taille résultante d'un découpage (slicing) avec un pas f.
- **Implémentation :** N'utilisez pas de fonctions toutes faites issues de bibliothèques de traitement d'images (comme OpenCV ou PIL) pour le redimensionnement. Implémentez la logique de sélection des pixels manuellement ou via le découpage de matrices.
- **Aliasing :** Notez que ce processus peut provoquer l'effet d'*aliasing* (crénelage), où les détails fins sont perdus ou des motifs indésirables apparaissent.

2.12.2.1 🧠 Discrétisation de l'espace

Le sous-échantillonnage réduit la résolution spatiale d'une image en ne sélectionnant qu'un pixel tous les f pixels dans chaque direction. C'est le processus inverse de l'interpolation :

Paramètre	Fonction	Effet
Facteur f	Saut d'échantillonnage	Définit l'intervalle de sélection. Un facteur 2 réduit la largeur et la hauteur de moitié.
Résolution	Densité de pixels	Diminue la quantité totale d'informations spatiales de l'image.
Aliasing	Effet secondaire	Apparition de motifs en escalier ou en blocs en raison de la perte de détails fins.

2.12.2.2 📋 Tâche (spécification pour VPL)

Entrée :

La première ligne contient `L`.

La deuxième ligne contient `C`.

La troisième ligne contient le facteur `f`.

Les lignes suivantes contiennent les éléments de la matrice $L \times C$.

Sortie :

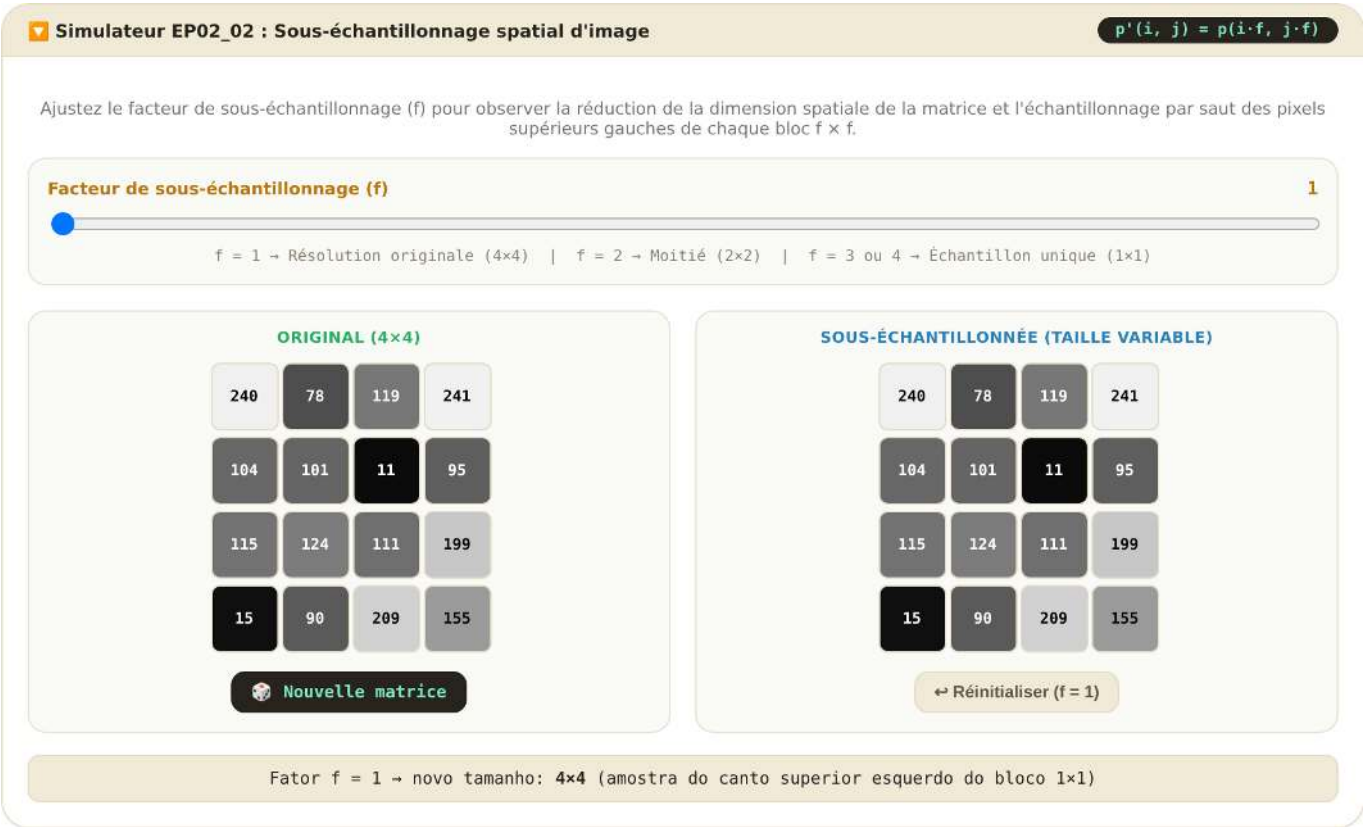
La matrice réduite avec les dimensions correspondant au découpage par `f`.

2.12.2.3 📌 Exemples

Entrée	Sortie	Observation
2 4 2 10 20 30 40 50 60 70 80	10 30	Le facteur 2 sélectionne les pixels (0,0) et (0,2) de la première ligne. La deuxième ligne est ignorée.

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

```
Overwriting EP02_02.py
```



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

Figure 2.13: Simulateur EP02_02: Sous-échantillonnage spatial (réduction de résolution par saut f)

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

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

2.12.3 EP02_03 🎨 Quantification des niveaux de gris

Dans cette activité, vous devez implémenter la quantification uniforme d'une image, en réduisant le nombre de niveaux d'intensité de gris originaux vers une nouvelle échelle basée sur un nombre de bits plus petit.

- Lisez deux entiers **L** et **C**, représentant les dimensions de la matrice.
- Lisez un entier k ($1 \leq k \leq 8$), représentant le nouveau nombre de bits de l'image.
- Calculez le nombre de niveaux ($N = 2^k$) et la taille de l'intervalle (pas).
- Pour chaque pixel p, calculez la nouvelle valeur p' en le mappant vers l'indice du niveau discrétisé correspondant (variant de 0 à $2^k - 1$).
- Affichez la matrice résultante avec les mêmes valeurs de dimensions que l'originale.
- Consultez Figure 2.14 pour une simulation de cet EP.

📌 **Important** :

- **Postérisation :** En réduisant drastiquement les niveaux (ex : $k = 2$), vous remarquerez que les dégradés lisses se transforment en bandes abruptes de couleur en raison de la perte de résolution d'amplitude.
- **Calcul du pas :** L'intervalle entre chaque niveau est défini par $pas = 256/2^k$.
- **Mappage :** La méthode de quantification uniforme par troncature qui mappe le pixel vers l'indice de son niveau discrétisé respectif est donnée par :

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

En termes d'implémentation (comme en Python), cela équivaut à la division entière : `p' = p // pas`.

2.12.3.1 🧠 Discrétisation de l'amplitude

Alors que le sous-échantillonnage traite de la résolution spatiale, la quantification se concentre sur la précision de la couleur (amplitude). Réduire les bits signifie simplifier l'information chromatique :

Paramètre	Fonction	Effet
Bits (k)	Profondeur de couleur	Définit combien de tons différents l'image peut avoir (2^k).
Pas	Intervalle de ton	Espacement entre les niveaux de gris autorisés.
Postérisation	Phénomène visuel	Transformation de variations continues en blocs de couleur unie.

2.12.3.2 📋 Tâche (spécification pour VPL)

Entrée :

La première ligne contient `L`.

La deuxième ligne contient `C`.

La troisième ligne contient le nombre de bits `k`.

Les lignes suivantes contiennent les éléments de la matrice $L \times C$.

Sortie :

La matrice transformée avec les indices des niveaux quantifiés, en conservant la taille originale $L \times C$.

2.12.3.3 📌 Exemples

Entrée	Sortie	Observation
1 4 2 0 80 170 255	0 1 2 3	Avec $k = 2$, nous avons $2^2 = 4$ niveaux discrets disponibles $(0, 1, 2, 3)$. Le pas est de $256/4 = 64$. En appliquant la division entière élément par élément : $0//64 = 0$, $80//64 = 1$, $170//64 = 2$, $255//64 = 3$.
1 5 1 10 50 120 200 250	0 0 0 1 1	Avec $k = 1$, nous avons $2^1 = 2$ niveaux (0 et 1). Pas $= 256/2 = 128$. Les pixels inférieurs à 128 donnent 0, et les pixels supérieurs ou égaux à 128 donnent 1.

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

```
Overwriting EP02_03.py
```

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

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

Figure 2.14: Simulateur EP02_03 : Quantification et Profondeur de Bits (Réduction du Nombre de Niveaux de Gris)

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

2.12.4 EP02_04 📐 Transformée de distance sur image binaire

Étant donné une image binaire où les pixels de valeur **1** représentent l'*objet* et les pixels **0** représentent le *fond*, la distance d'un pixel de fond reçoit la **distance minimale au pixel d'objet le plus proche**. Les pixels d'objet reçoivent une distance **0**. Pour simplifier, considérez que l'image ne contient **qu'un seul objet avec un seul pixel de valeur 1**.

Problème : Lire une image binaire $L \times C$ et une métrique, puis calculer cette distance simplifiée en appliquant l'une des trois formules :

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

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

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

où Δr est la différence de lignes et Δc la différence de colonnes entre deux pixels.

2.12.4.1 🖼️ Pourquoi cela importe-t-il ? - Applications de la TD

La Transformée de distance (TD) apparaît dans des dizaines de pipelines de vision par ordinateur :

Métrique	Complexité	Application typique
Euclidienne	🔴 $O(n^2)$ naïve	Squelettisation, correspondance de formes
City-block	🟡 $O(n)$ avec 2 passes	Morphologie, dilatation/érosion
Échiquier	🟢 $O(n)$ avec 2 passes	Morphologie, dilatation/érosion

2.12.4.2 📌 Exigences techniques

- **Entrée :** * Première ligne : L et C (entiers).
  - Deuxième ligne : nom de la métrique (`euclidean`, `cityblock` ou `chessboard`).
  - Ensuite, la matrice binaire $L \times C$ (valeurs 0 ou 1).
- **Pixels d'objet (1) :** distance = 0 (ou 0.00 pour euclidienne).
- **Pixels de fond (0) :** distance au seul pixel d'objet de l'image.
- **Arrondi (euclidienne) :** imprimer avec **2 décimales** (format `:.2f`). City-block et Échiquier produisent des entiers — imprimer sans décimales.
- **Sortie :** valeurs séparées par des espaces, une ligne par ligne de la matrice.
- Voir dans Figure 2.15 une simulation de cet EP.

2.12.4.3 📌 Exemples

Entrée	Sortie	Observation
4 4 chessboard 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0	2 2 2 2 2 1 1 1 2 1 0 1 2 1 1 1	La distance Échiquier est $\max(\|dx\|, \|dy\|)$. Le seul pixel objet est $(2, 2) = 0$; les autres stockent leur distance minimale jusqu'à lui.

2.12.4.4 📌 Observations finales

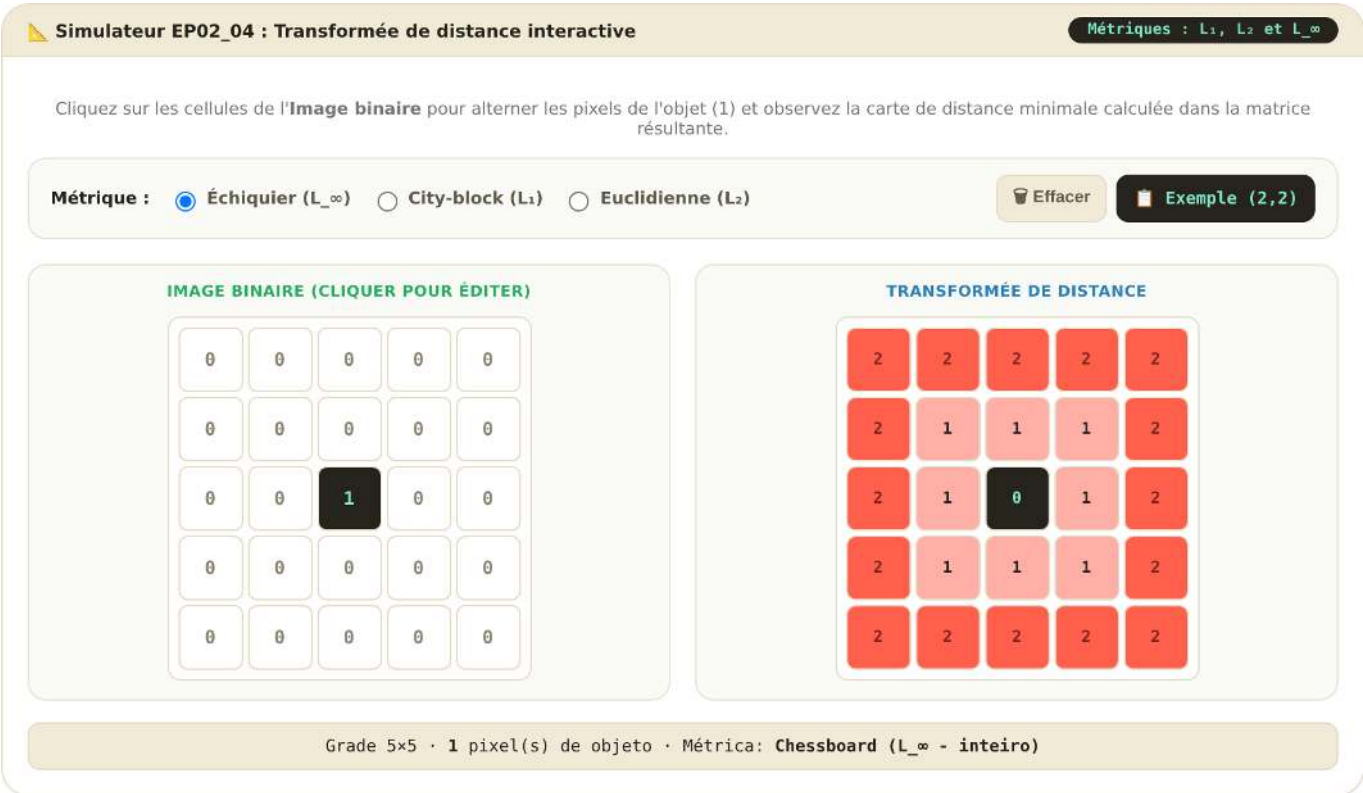
- Comme l'image ne contient **qu'un seul objet d'un pixel**, la distance de chaque pixel de fond est simplement la distance de ce pixel au seul point objet.
- L'implémentation peut utiliser la force brute (parcourir tous les pixels de l'image et calculer la distance directement), car L et C sont petits dans les cas de test.
- Ce problème est un échauffement pour la Transformée de distance générale, qui sera travaillée dans des chapitres ultérieurs avec plusieurs objets et des algorithmes optimisés.

```
%%writefile EP02_04.py
# Code 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.fr/cap02/sim-ep0204-distancia.html

Figure 2.15: Simulateur EP02_04 : Transformée de distance en image binaire (Chessboard, City-block et Euclidienne)

2.12.5 EP02_05 Translation d'image

Dans cette activité, vous devez implémenter le déplacement spatial d'une image. La translation déplace chaque pixel de l'image originale vers une nouvelle position en fonction d'un vecteur de déplacement.

- Lisez deux entiers **L** et **C**, représentant les dimensions de la matrice.
- Lisez deux entiers t_x (déplacement horizontal) et t_y (déplacement vertical).
- Lisez les valeurs entières de la matrice originale.
- Calculez la nouvelle position (x', y') pour chaque pixel (x, y) original.
- Affichez la matrice résultante avec les mêmes dimensions que l'originale.
- Voir dans Figure 2.16 une simulation de cet EP.

📌 **Important** :

- **Remplissage :** Les pixels qui « entrent » dans l'image en raison du déplacement et qui n'ont pas de correspondant dans l'originale doivent être remplis avec **0** (noir).
- **Suppression :** Les pixels qui, après la translation, sortent des limites de la matrice ($0 \dots L-1$ ou $0 \dots C-1$) doivent être ignorés.
- **Coordonnées :** Considérez x comme l'indice de la ligne et y comme l'indice de la colonne.

2.12.5.1 🧠 Déplacement spatial

Translater une image signifie déplacer tous ses points d'une distance fixe dans des directions spécifiées. Mathématiquement, en utilisant les coordonnées homogènes, l'opération est décrite comme suit :

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

Ce qui donne les équations simples :

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

2.12.5.2 📋 Tâche (spécification pour VPL)

Entrée :

La première ligne contient `L`.

La deuxième ligne contient `C`.

La troisième ligne contient les entiers `tx` et `ty`.

Les lignes suivantes contiennent les éléments de la matrice $L \times C$.

Sortie :

La matrice résultante avec les mêmes dimensions $L \times C$ après le déplacement.

2.12.5.3 📌 Exemples

Entrée	Sortie	Observation
2 2 1 1 10 20 30 40	0 0 0 10	**Déplacement ($t_x = 1, t_y = 1$):** Chaque pixel se déplace d'une position vers la droite (horizontal) et d'une vers le bas (vertical). Le pixel $(0, 0) = 10$ va vers la destination $(1, 1)$ (coin inférieur droit). Les positions vides sont remplies avec 0.
3 3 -1 0 1 2 3 4 5 6 7 8 9	2 3 0 5 6 0 8 9 0	**Déplacement ($t_x = -1, t_y = 0$):** Chaque pixel se déplace d'une position vers la gauche (horizontal). La première colonne originale (`1`, `4`, `7`) est supprimée, les autres colonnes se déplacent vers la gauche, et la dernière colonne résultante est remplie de zéros (0).

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

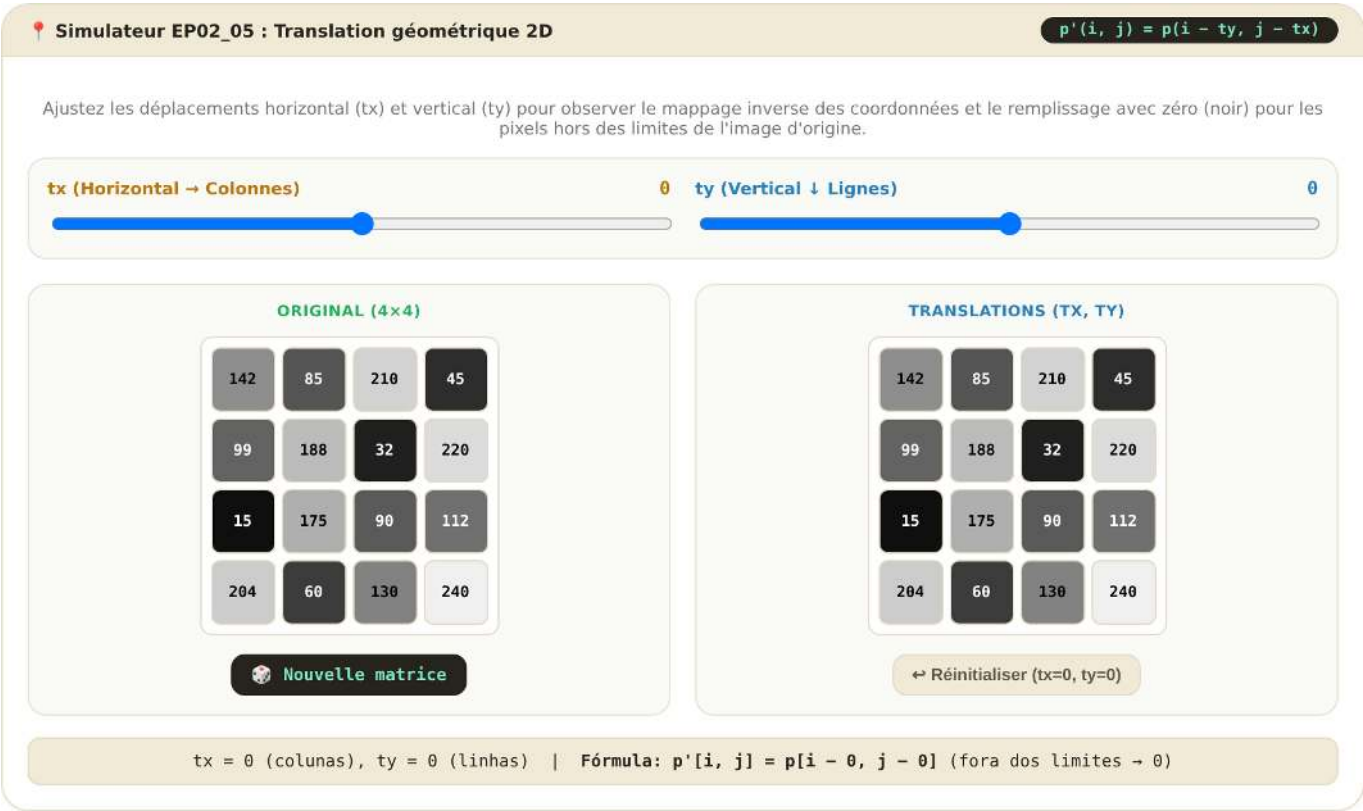
```
Overwriting EP02_05.py
```

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

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

2.12.6 EP02_06 🔄 Rotation d'image

Dans cette activité, vous devez implémenter la rotation d'une image autour de son centre géométrique. Cette opération nécessite le mappage des coordonnées et l'utilisation de techniques d'interpolation pour déterminer les nouvelles valeurs des pixels.



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

Figure 2.16: Simulateur EP02_05 : Translation géométrique d'image (Déplacement tx et ty avec remplissage de bordure)

- Lisez deux entiers **L** et **C**, représentant les dimensions de la matrice.
- Lisez une valeur réelle θ (angle en degrés) et une *chaîne de caractères* représentant la **méthode** d'interpolation (`nearest` ou `bilinear`).
- Lisez les valeurs entières de la matrice originale.
- Effectuez la rotation autour du centre de l'image $(L/2, C/2)$.
- Affichez la matrice résultante avec les mêmes dimensions que l'originale.
- Voir dans Figure 2.17 une simulation de cet EP.

📌 **Important** :

- **Mappage inverse :** Pour éviter les « trous » dans l'image finale, parcourez chaque pixel (x', y') de l'image de destination et calculez sa position correspondante (x, y) dans l'image originale en utilisant la matrice de rotation inverse.
- **Interpolation :**
- `nearest` : Attribue la valeur du pixel le plus proche de la coordonnée calculée.
- `bilinear` : Calcule une moyenne pondérée basée sur les 4 voisins les plus proches.
- **Bords :** Les pixels dont l'origine (x, y) tombe en dehors des limites de l'image originale doivent être remplis avec **0**.

2.12.6.1 🧠 Transformation par angle

La rotation d'un point (x, y) par rapport à l'origine d'un angle θ est donnée par la matrice de transformation. Pour effectuer une rotation autour d'un centre (x_c, y_c), on translate d'abord le centre vers l'origine, on effectue la rotation, puis on translate en retour :

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

Conseil : Utilisez le mappage inverse pour garantir que tous les pixels de l'image de sortie soient correctement remplis.

2.12.6.2 📋 Tâche (spécification pour VPL)

Entrée :

La première ligne contient `L`.

La deuxième ligne contient `C`.

La troisième ligne contient l'angle `theta` (en degrés) et la méthode `interp` (`nearest` ou `bilinear`).

Les lignes suivantes contiennent les éléments de la matrice $L \times C$.

Sortie :

La matrice tournée avec `L` lignes et `C` colonnes.

2.12.6.3 📌 Exemples

Entrée	Sortie	Observation
2 2 90 nearest 1 2 3 4	3 1 4 2	Rotation de 90° dans le sens horaire : la colonne 0 devient la ligne 0 (de bas en haut). $(0,0) = 1 \to (1,0)$, $(1,0) = 3 \to (0,0)$, $(0,1) = 2 \to (1,1)$, $(1,1) = 4 \to (0,1)$.
3 3 45 bilinear 0 0 0 0 255 0 0 0 0	0 180 0 180 255 180 0 180 0	Rotation de 45° : le pixel central reste 255 ; les voisins directs reçoivent une valeur interpolée ≈ 180 par bilinéaire ; les coins restent à 0.

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

```
Overwriting EP02_06.py
```

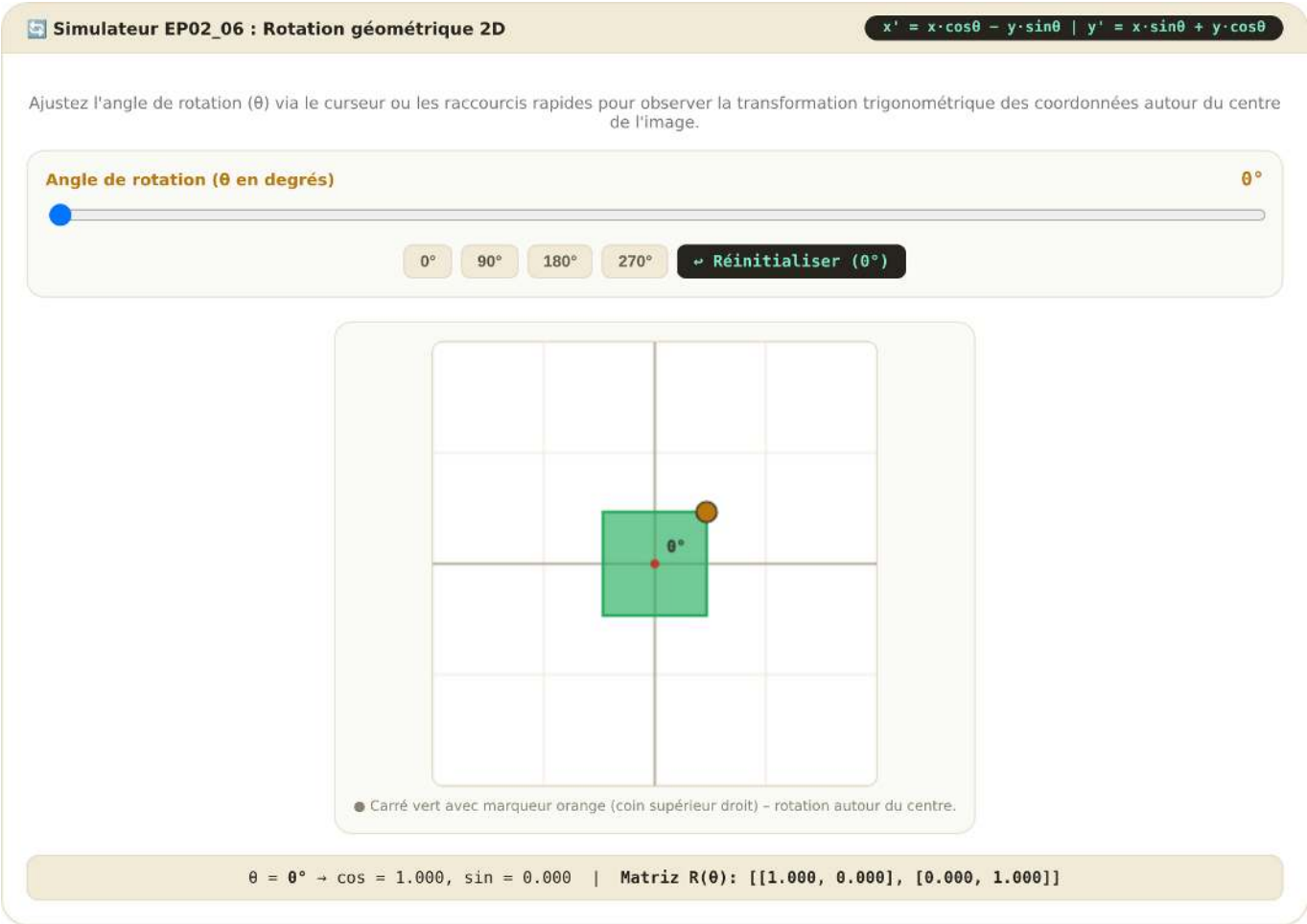
```
TestSuite("EP02_06.py").run()
```

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

2.12.7 EP02_07 🔍 Redimensionnement (Échelle)

Dans cette activité, vous devez implémenter le redimensionnement d'une image à l'aide de facteurs d'échelle. Contrairement au sous-échantillonnage simple, nous utiliserons ici des techniques d'interpolation pour permettre à la fois l'agrandissement et la réduction de l'image.

- Lisez deux entiers **L** et **C**, représentant les dimensions de la matrice originale.



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

Figure 2.17: Simulateur EP02_06 : Rotation d'image autour de l'origine par un angle

- Lisez deux valeurs réelles s_x (échelle sur les lignes) et s_y (échelle sur les colonnes).
- Lisez une *chaîne de caractères* représentant la **méthode** d'interpolation (`nearest` ou `bilinear`).
- Lisez les valeurs entières de la matrice originale.
- Calculez les nouvelles dimensions : $L' = \text{round}(L \times s_x)$ et $C' = \text{round}(C \times s_y)$.
- Affichez la matrice résultante avec les nouvelles dimensions.
- Voir dans Figure 2.18 une simulation de cet EP.

📌 **Important** :

- **Mappage inverse :** Pour chaque pixel (x', y') de l'image de destination, trouvez la position correspondante dans l'origine en utilisant $(x, y) = (x'/s_x, y'/s_y)$.
- **Interpolation :**
- `nearest` : Sélectionne la valeur du pixel le plus proche (arrondi des coordonnées).
- `bilinear` : Effectue une interpolation linéaire double entre les quatre pixels voisins les plus proches dans l'image originale.
- **Bords :** Assurez-vous que le mappage ne tente pas d'accéder à des indices hors de l'intervalle $[0, L-1]$ et $[0, C-1]$.

2.12.7.1 🧠 Interpolation pour Agrandissement/Réduction

Redimensionner une image par des facteurs (s_x, s_y) exige le remplissage des vides (lors de l'agrandissement) ou la fusion d'informations (lors de la réduction). La méthode d'interpolation définit la qualité visuelle du résultat :

Méthode	Fonctionnement	Effet visuel
Nearest	Prend la valeur du voisin le plus proche.	Rapide, mais génère un effet "pixelisé" ou des blocs.
Bilinear	Moyenne pondérée des 4 voisins (2×2).	Adoucit l'image, réduisant le crénelage.

2.12.7.2 📋 Tâche (spécification pour VPL)

Entrée :

La première ligne contient `L`.

La deuxième ligne contient `C`.

La troisième ligne contient les facteurs `sx` et `sy`.

La quatrième ligne contient la méthode `interp` (`nearest` ou `bilinear`).

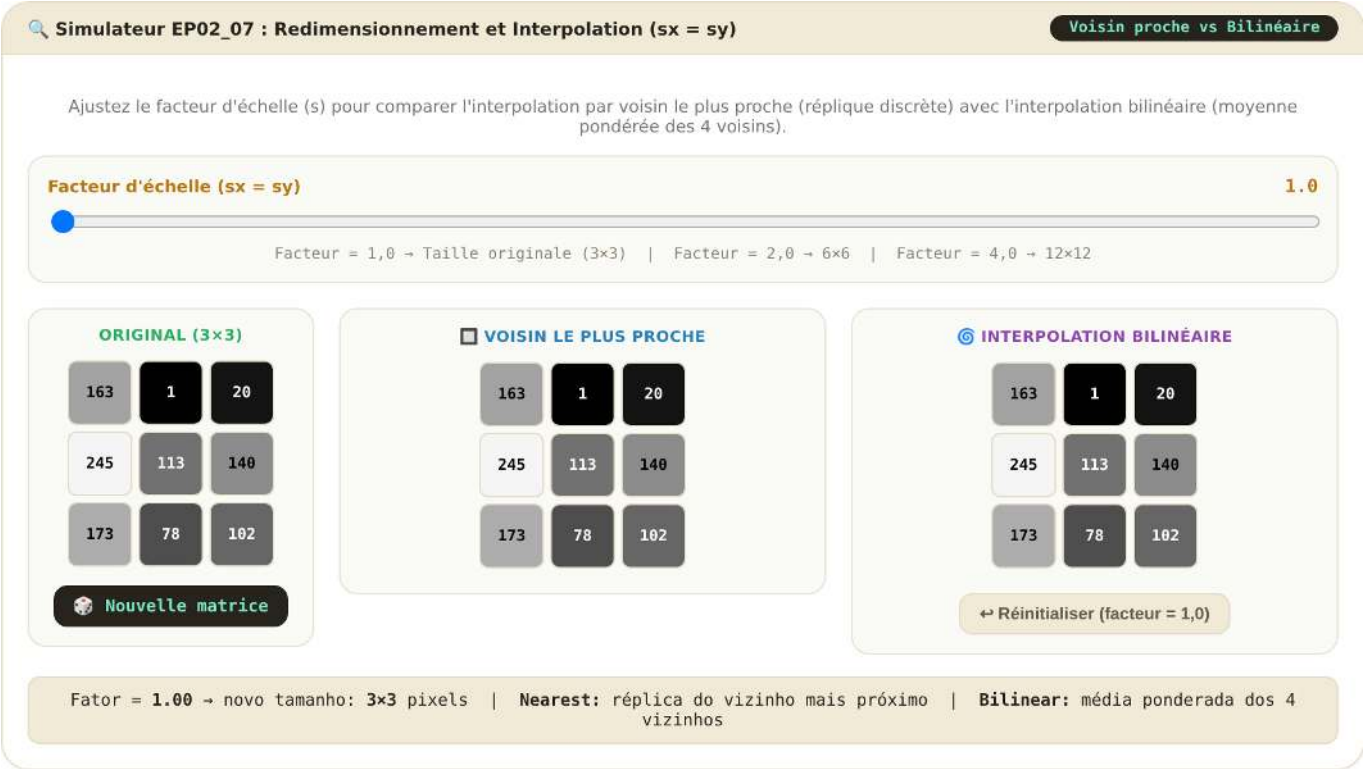
Les lignes suivantes contiennent les éléments de la matrice $L \times C$.

Sortie :

La matrice redimensionnée avec les dimensions $L' \times C'$.

2.12.7.3 📌 Exemples

Entrée	Sortie	Observation
2 2 2.0 2.0 nearest 1 2 3 4	1 1 2 2 1 1 2 2 3 3 4 4 3 3 4 4	Agrandissement 2× : chaque pixel original est répliqué dans un bloc 2×2. L'image 2×2 devient 4×4.
2 2 0.5 0.5 nearest 10 20 30 40	10	Réduction 0.5× : l'image 2×2 devient 1×1. Avec nearest, le seul pixel de sortie échantillonne la position $(0, 0) = 10$.



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

Figure 2.18: Simulateur EP02_07 : Redimensionnement spatial et interpolation (Nearest Neighbor vs Bilinéaire)

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

# 1. Lecture des dimensions, facteurs et méthode
l = int(input())
c = int(input())
sx, sy = map(float, input().split())
interp = input().strip()

# 2. Lecture de l'image originale
img = mm.readImg(l, c)
```

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

# 4. Redimensionnement avec mm.resize
# cv2.resize utilise (largeur, hauteur) = (colonnes, lignes)
resultado = mm.resize(img, (c_new, l_new), method=interp)

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

```
Overwriting EP02_07.py
```

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

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

2.12.8 EP02_08 🔀 Cisaillement (Shear)

Dans cette activité, vous devez implémenter la transformation de cisaillement sur une image. Le cisaillement est une transformation affine qui décale chaque point dans une direction fixe, d'une valeur proportionnelle à sa distance par rapport à une droite parallèle à cette direction, produisant un effet d'inclinaison.

- Lire deux entiers **L** et **C**, représentant les dimensions de la matrice.
- Lire deux valeurs réelles sh_x (cisaillement horizontal) et sh_y (cisaillement vertical).
- Lire une *chaîne de caractères* représentant la **méthode** d'interpolation (`nearest` ou `bilinear`).
- Lire les valeurs entières de la matrice originale.
- Appliquer la transformation en conservant la taille originale de l'image (en coupant ce qui dépasse les limites).
- Afficher la matrice résultante avec les dimensions $L \times C$.
- Voir dans Figure 2.19 une simulation de cet EP.

📌 **Important** :

- **Mappage inverse :** Pour chaque pixel (x', y') de l'image de destination, calculer la position correspondante dans l'origine (x, y) en utilisant la matrice de cisaillement inverse.
- **Remplissage :** Les coordonnées qui aboutissent à des positions hors de la matrice originale doivent être remplies avec **0**.
- **Coordonnées :** Pour les besoins de cette implémentation, considérez x comme l'indice de ligne et y comme l'indice de colonne.

2.12.8.1 🧠 Distorsion affine

Le cisaillement modifie la géométrie de l'image en inclinant ses axes. La relation entre les coordonnées originales (x, y) et les coordonnées transformées (x', y') est donnée par :

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

Cela donne les équations :

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

2.12.8.2 📋 Tâche (spécification pour VPL)

Entrée :

La première ligne contient `L`.

La deuxième ligne contient `C`.

La troisième ligne contient les facteurs `shx` et `shy`.

La quatrième ligne contient la méthode `interp` (`nearest` ou `bilinear`).

Les lignes suivantes contiennent les éléments de la matrice $L \times C$.

Sortie :

La matrice transformée avec les mêmes dimensions $L \times C$.

2.12.8.3 📌 Exemples

Entrée	Sortie	Observation
3 3 0.5 0.0 nearest 10 20 30 40 50 60 70 80 90	10 20 30 0 40 50 0 0 70	Cisaillement horizontal : la ligne i se décale de $\lfloor i \cdot 0.5 \rfloor$ pixels. Ligne $0 \to 0$px, ligne $1 \to 0$px, ligne $2 \to 1$px. Les pixels décalés vers l'extérieur sont éliminés et les positions vides sont remplies avec 0.
2 2 0.0 1.0 nearest 10 20 30 40	10 0 30 20	Cisaillement vertical : la colonne j se décale de $\lfloor j \cdot 1.0 \rfloor$ pixels vers le bas. Colonne $0 \to 0$px (inchangée), colonne $1 \to 1$px : 20 descend à $(1, 1)$ et $(0, 1)$ devient 0.

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

```
Overwriting EP02_08.py
```

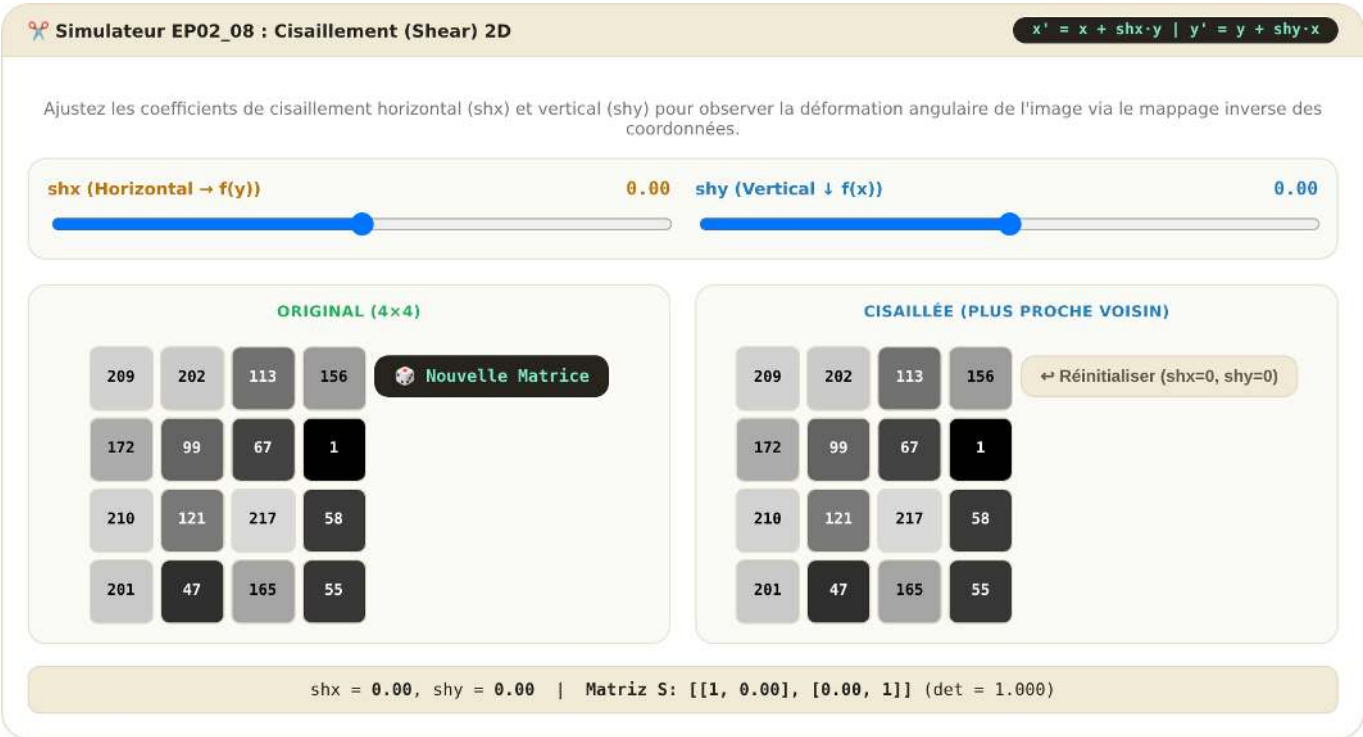
```
TestSuite("EP02_08.py").run()
```

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

2.12.9 EP02_09 🧩 Transformation Affine Générique

Dans cette activité, vous devez implémenter une transformation affine arbitraire sur une image. Cette opération est la généralisation de toutes les transformations linéaires (mise à l'échelle, rotation, cisaillement) combinées à la translation, permettant des manipulations géométriques complexes via une seule matrice.

- Lisez deux entiers **L** et **C**, représentant les dimensions de la matrice.
- Lisez **six valeurs réelles** (a, b, t_x, c, d, t_y) qui composent la matrice de transformation affine 2×3.
- Lisez une *chaîne de caractères* représentant la **méthode** d'interpolation (`nearest` ou `bilinear`).
- Lisez les valeurs entières de la matrice d'origine.
- Appliquez la transformation en conservant la taille d'origine $L \times C$.
- Affichez la matrice résultante.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap02/sim-ep0208-cisalhamento.html

Figure 2.19: Simulateur EP02_08 : Transformation géométrique de cisaillement 2D (Cisaillement horizontal et vertical)

- Voir Figure 2.20 pour une simulation de cet EP.

📌 **Important** :

- **Mappage inverse :** Pour calculer la valeur de chaque pixel dans l'image de destination, vous devez utiliser l'**inverse** de la matrice de transformation affine fournie afin de trouver la coordonnée correspondante dans l'image d'origine.
- **Remplissage :** Les coordonnées calculées qui tombent hors des limites $[0, L-1]$ et $[0, C-1]$ de l'image d'origine doivent donner un pixel de valeur **0**.
- **Flexibilité :** Cette implémentation doit être capable d'effectuer n'importe laquelle des tâches précédentes (translation, rotation, etc.) en modifiant simplement les paramètres de la matrice.

Astuce :

```
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 🧠 Combinaison d'opérations

La transformation affine préserve les points, les droites et les plans. En traitement d'images, elle mappe la position (x, y) vers (x', y') selon le système :

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

Ou, de manière compacte en coordonnées homogènes :

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

2.12.9.2 📋 Tâche (spécification pour VPL)

Entrée :

La première ligne contient `L`.

La deuxième ligne contient `C`.

La troisième ligne contient six flottants : `a b tx c d ty`.

La quatrième ligne contient la méthode `interp` (`nearest` ou `bilinear`).

Les lignes suivantes contiennent les éléments de la matrice $L \times C$.

Sortie :

La matrice transformée avec les dimensions d'origine $L \times C$.

2.12.9.3 📌 Exemples

Entrée	Sortie	Observation
2 2 1.0 0.0 0.5 0.0 1.0 0.5 bilinear 10 20 30 40	15 20 25 30	Translation fractionnaire ($t_x = 0.5, t_y = 0.5$) : chaque pixel de sortie (i, j) échantillonne la position $(i + 0.5,\ j + 0.5)$ de l'entrée via bilinéaire. Ex : $(0, 0)$ interpole les quatre voisins $\rightarrow$ 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	Mise à l'échelle 2× via matrice affine ($a = 2, d = 2$) : chaque pixel de sortie (i, j) échantillonne la position $(2i, 2j)$ de l'entrée avec le plus proche voisin. Ex : $(0, 2) \rightarrow (0, 4)$ hors de l'image $\rightarrow$ le plus proche voisin découpe vers $(0, 2) = 3$… en attente de confirmation de la logique de bord.

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

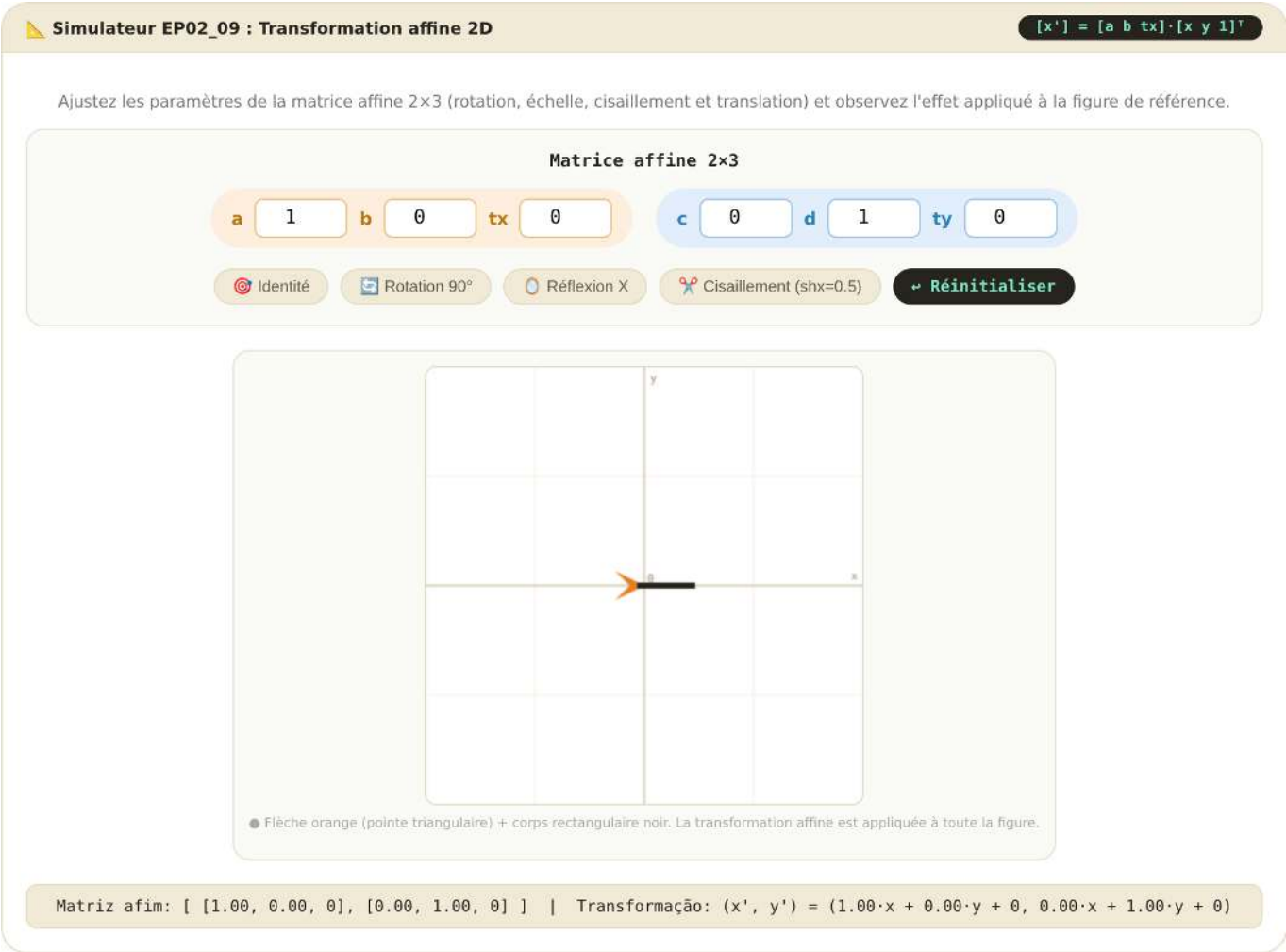
```
Overwriting EP02_09.py
```

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

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

2.12.10 EP02_10 🎯 Correction de perspective (homographie)

Dans cette activité, vous devez implémenter la transformation de perspective, également connue sous le nom d'homographie. Contrairement aux transformations affines, la perspective ne préserve pas le parallélisme, ce



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap02/sim-ep0209-afim.html

Figure 2.20: Simulador EP02_09 : Transformação Afim 2D (Matriz 2×3)

qui permet de « redresser » des objets inclinés, comme des documents ou des panneaux capturés sous des angles obliques.

- Lisez deux entiers **L** et **C**, représentant les dimensions de la matrice d'origine.
- Lisez **quatre paires de coordonnées** (x, y) représentant les coins du quadrilatère source (objet déformé).
- Lisez **quatre paires de coordonnées** (x, y) représentant les coins du quadrilatère de destination (là où l'objet doit être mappé).
- Lisez les valeurs de la matrice d'origine.
- Calculez la matrice d'homographie 3×3 et appliquez la transformation.
- Affichez la matrice résultante avec les dimensions de sortie spécifiées.
- Voir dans Figure 2.21 une simulation de cet EP.

📌 **Important** :

- **Degrés de liberté :** L'homographie possède 8 degrés de liberté (le neuvième élément de la matrice 3×3 est une constante de normalisation, généralement 1), nécessitant au minimum 4 points correspondants pour être calculée.
- **Projection :** Après avoir multiplié les coordonnées par la matrice, il est nécessaire de diviser les résultats x' et y' par la composante homogène w pour revenir au plan 2D.
- **Utilisation de bibliothèques :** Pour cette tâche, vous pouvez utiliser les fonctions `cv2.getPerspectiveTransfo` pour obtenir la matrice et `cv2.warpPerspective` pour appliquer la transformation, ou implémenter le système linéaire et le mappage inverse manuellement pour un défi supplémentaire.

```
# Dimensions de sortie : boîte englobante des points de destination + 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))
# ou
dst = mm.perspective_transform(img, pts1, pts2, size=(w, h))
```

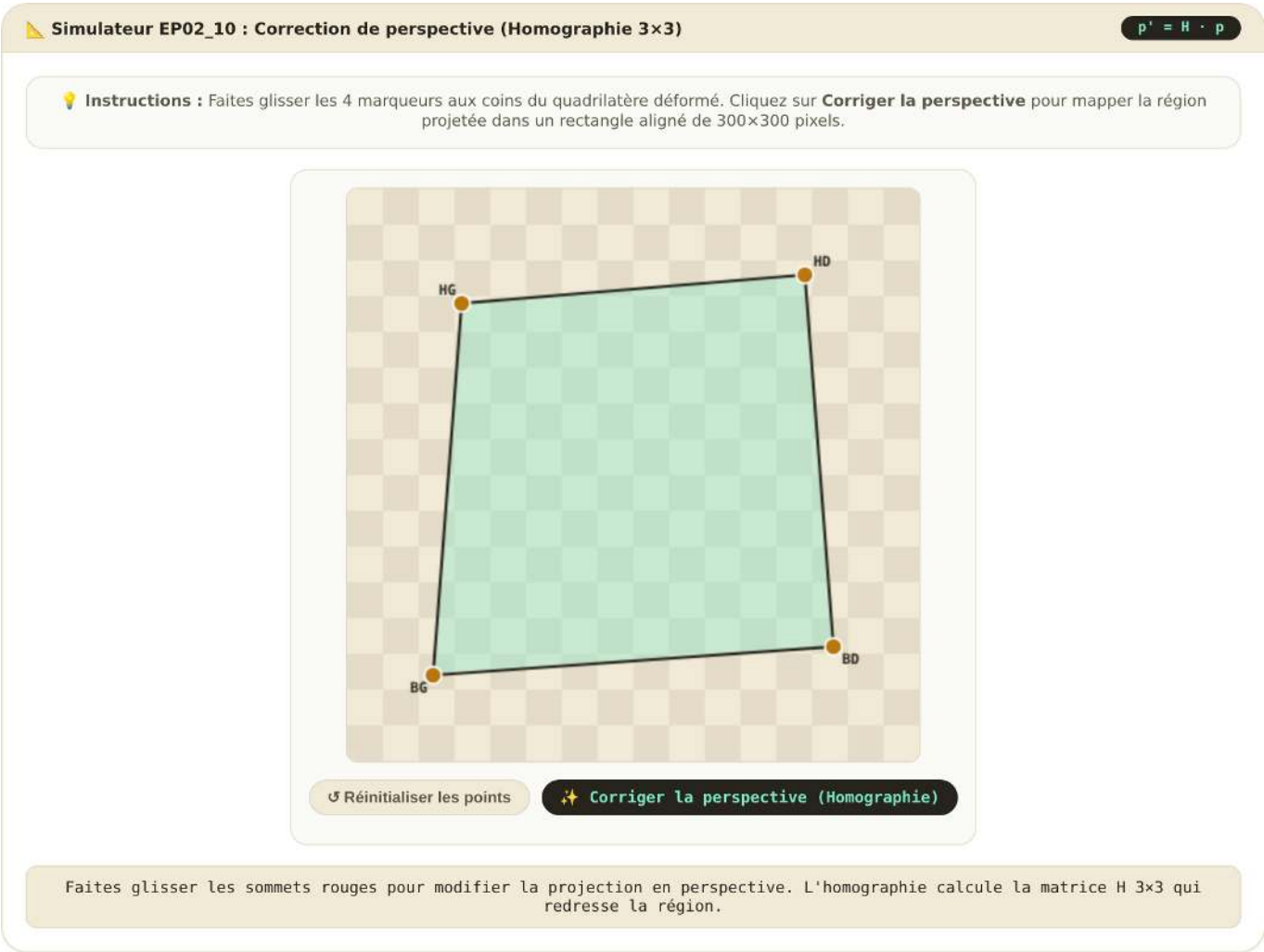
2.12.10.1 🧠 Déformation non affine

Tandis que les transformations affines mappent des parallélogrammes en parallélogrammes, l'homographie mappe tout quadrilatère en un autre quadrilatère. Cela est essentiel pour la vision par ordinateur :

Opération	Caractéristique	Application typique
Homographie	Projection sur un plan	Correction de documents, numérisation de panneaux.
Point de fuite	Convergence de lignes	Reconstruction 3D à partir d'images 2D.
Warping	Déformation de maillage	Stabilisation vidéo et panoramas (stitching).

2.12.10.2 📌 Exemples

Entrée	Sortie	Observation
4 4 0 0 3 0 0 3 3 3 0 0 3 0 0 3 3 3 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160	10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160	Les 4 premières lignes après les dimensions sont les points sources ; les 4 suivantes sont les destinations. Avec des points identiques, la transformation de perspective est l'identité et l'image est préservée.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap02/sim-ep0210-perspectiva.html

Figure 2.21: Simulateur EP02_10 : Correction de perspective (Transformation d'homographie 3×3)

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

```
Overwriting EP02_10.py
```

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

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

2.12.11 EP02_11 🏆 Correction de Perspective (Homographie) sur une Image Réelle

Dans cette activité, l'objectif est d'appliquer la transformation de perspective (homographie) pour « redresser » un objet incliné sur une photographie réelle. Vous travaillerez avec l'image d'un journal, où la grille d'un jeu de Sudoku est déformée en raison de l'angle sous lequel la photo a été prise.

Votre programme doit lire les paramètres d'entrée depuis le terminal, charger l'image, calculer la matrice d'homographie 3×3, appliquer la transformation géométrique et afficher un indicateur global de validation.

- Lisez deux entiers **L** et **C**, représentant les dimensions en lignes et en colonnes (hauteur et largeur) que l'image redressée de sortie doit avoir.
- Lisez **quatre paires de coordonnées** (x, y) via le terminal, représentant les quatre coins du quadrilatère d'origine (le Sudoku déformé dans l'image originale).
- Calculez automatiquement les **quatre paires de coordonnées de destination** en utilisant les dimensions L et C fournies, en mappant les coins vers les extrémités de la nouvelle image : $(0, 0)$, $(C-1, 0)$, $(0, L-1)$ et $(C-1, L-1)$.
- Chargez l'image locale `sudoku.png` et convertissez-la en niveaux de gris (grayscale).
- Calculez la matrice d'homographie et appliquez la transformation spatiale sur l'image.
- **Sortie :** Calculez et imprimez la **somme de tous les pixels** de l'image résultante.

📌 **Important** :

- **Fichier d'entrée :** L'image `sudoku.png` doit se trouver dans le même dossier que le script. Le programme doit la lire directement depuis le disque (ex : en utilisant `mm.read("sudoku.png")` ou `cv2.imread`).
- **Ordre des points :** Assurez-vous que la lecture des 4 points d'origine et la génération des 4 points de destination suivent rigoureusement le même ordre des coins : Supérieur-Gauche (TL), Supérieur-Droit (TR), Inférieur-Gauche (BL) et Inférieur-Droit (BR).
- **Dimensions dans OpenCV :** N'oubliez pas que des fonctions comme `cv2.warpPerspective` attendent la taille de l'image de sortie au format `(largeur, hauteur)`, ce qui équivaut à `(C, L)`.
- **Interpolation :** Pour garantir la cohérence mathématique de la somme des pixels avec le correcteur automatique, utilisez l'interpolation bilinéaire par défaut (`flags=cv2.INTER_LINEAR`).
- **Crédits :** L'image utilisée est **« Sudoku en periódico »** de Héctor Rodríguez, sous licence **CC BY 2.0**.

2.12.11.1 🧠 Contexte du Problème

L'homographie possède 8 degrés de liberté, nécessitant au minimum 4 correspondances de points pour être calculée. Contrairement aux transformations affines, elle mappe n'importe quel quadrilatère sur un autre quadrilatère, permettant ainsi aux lignes qui convergent vers des points de fuite de redevenir parallèles :

Opération	Caractéristique	Application Typique
Homographie	Projection entre plans	Rectification de documents, numérisation de plaques et de codes QR.
Mappage inverse	Balayage de la destination vers l'origine	Évite les « trous » ou les pixels vides dans l'image finale redressée.
Warping	Rééchantillonnage spatial	Correction de la distorsion des lentilles et assemblage de panoramas (*stitching*).

2.12.11.2 📌 Exemples

Entrée	Sortie	Observation
500 500 100 120 420 95 80 440 450 460	32982820	Les deux premières entrées sont les dimensions de sortie (L et C). Les 4 lignes suivantes sont les coordonnées (x, y) des coins du Sudoku dans l'image originale + PAD. La sortie est la somme totale des pixels de l'image redressée.
200 200 100 120 420 95 80 440 450 460	5277150	Mêmes points d'origine que l'exemple précédent, mais en générant une image de sortie plus petite (200×200). La somme des pixels diminue proportionnellement en raison de l'échelle.

2.12.11.3 Acquisition de l'image du sudoku et conversion en niveaux de gris

La Figure 2.22 montre la lecture de l'image originale, suivie de la conversion en tons de gris et du redimensionnement en une matrice de 500×500 pixels, préparant les données pour l'étape suivante.

La correction de perspective, appliquée via la matrice d'homographie dans Figure 2.23, élimine les déformations causées par l'angle de la caméra et produit une vue frontale et régulière de la grille de Sudoku.

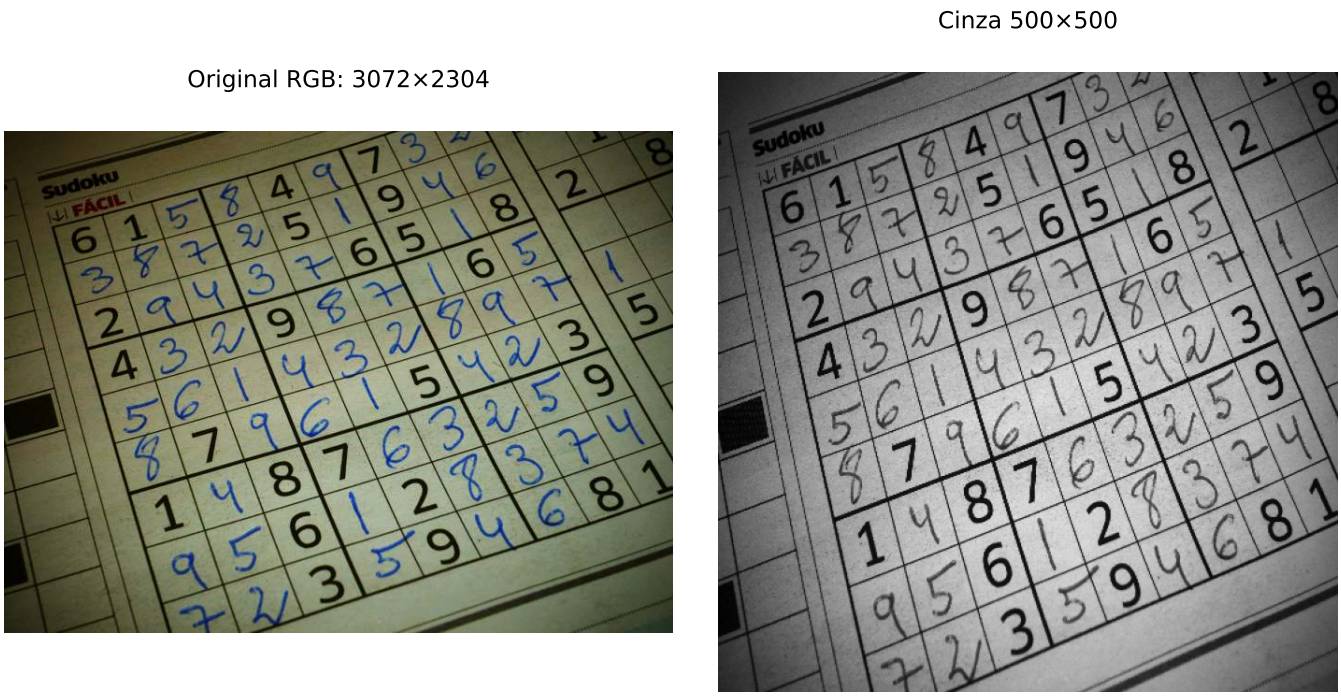


Figure 2.22: Aquisição da imagem de um Sudoku à esquerda. À direita, conversão para tons de cinza e redimensionamento. Crédito: Héctor Rodríguez de Guardamar, Espanha (CC BY 2.0).

```
import cv2
import numpy as np
```

```python
# --- 1. Charger l'image enregistrée (sudoku.png) ---
img = mm.read("sudoku.png")                # BGR, 500×500

# --- 2. Padding pour ne pas couper les sommets ---
PAD = 60
img_pad = cv2.copyMakeBorder(
    img, PAD, PAD, PAD, PAD,
    cv2.BORDER_CONSTANT, value=[255, 255, 255]
)

# --- 3. Points d'origine (coins de la grille dans l'image agrandie) ---
pts1 = np.float32([
    [100, 160],    # TL
    [390, 45],    # TR
    [200, 580],   # BL
    [570, 420],   # BR
])
#     W    H

# --- 4. Points de destination (vue frontale 500×500) ---
SIZE = 500
pts2 = np.float32([
    [0,    0],
    [SIZE, 0],
    [0,    SIZE],
    [SIZE, SIZE],
])

# --- 5. Homographie et redressement ---
img_rect = mm.perspective_transform(img_pad, pts1, pts2, size=(SIZE, SIZE))

# --- 6. Affichage ---
mm.show(
    [img_pad, img_rect],
    titles=["Original (avec padding)", "Vue frontale redressée"],
    cols=2, figsize=(10, 6), axis=True
)
```

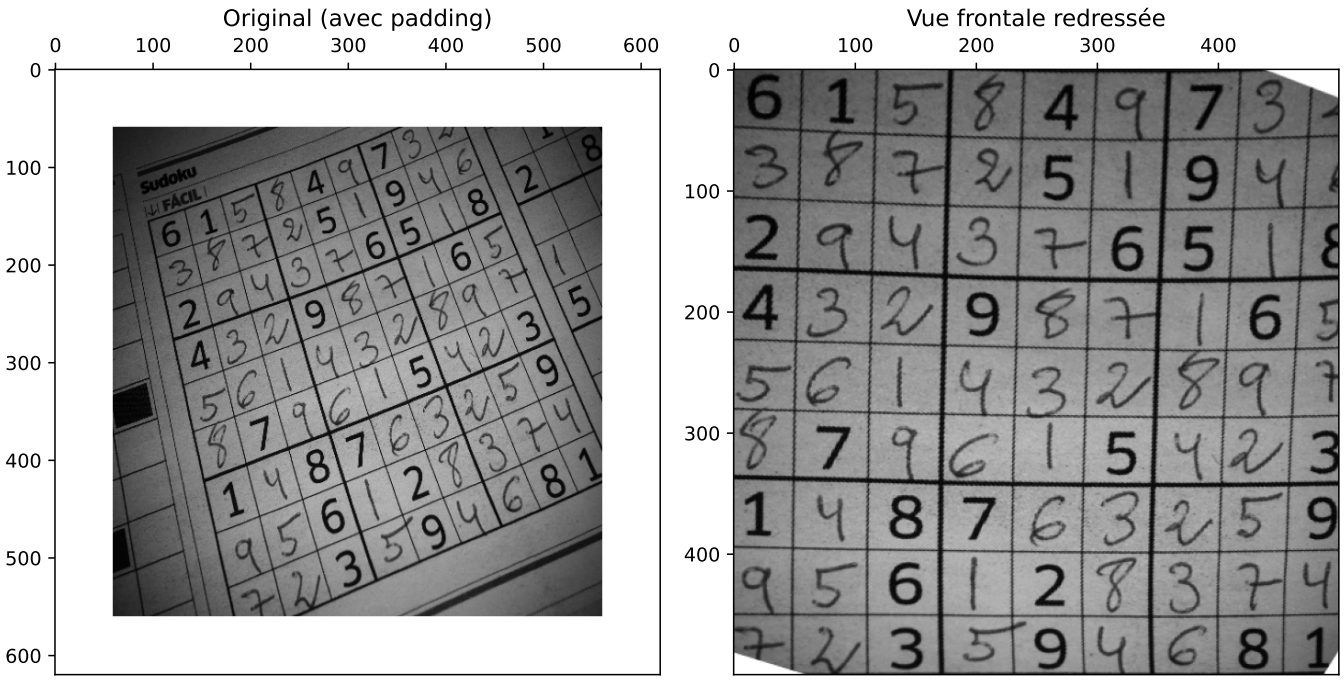
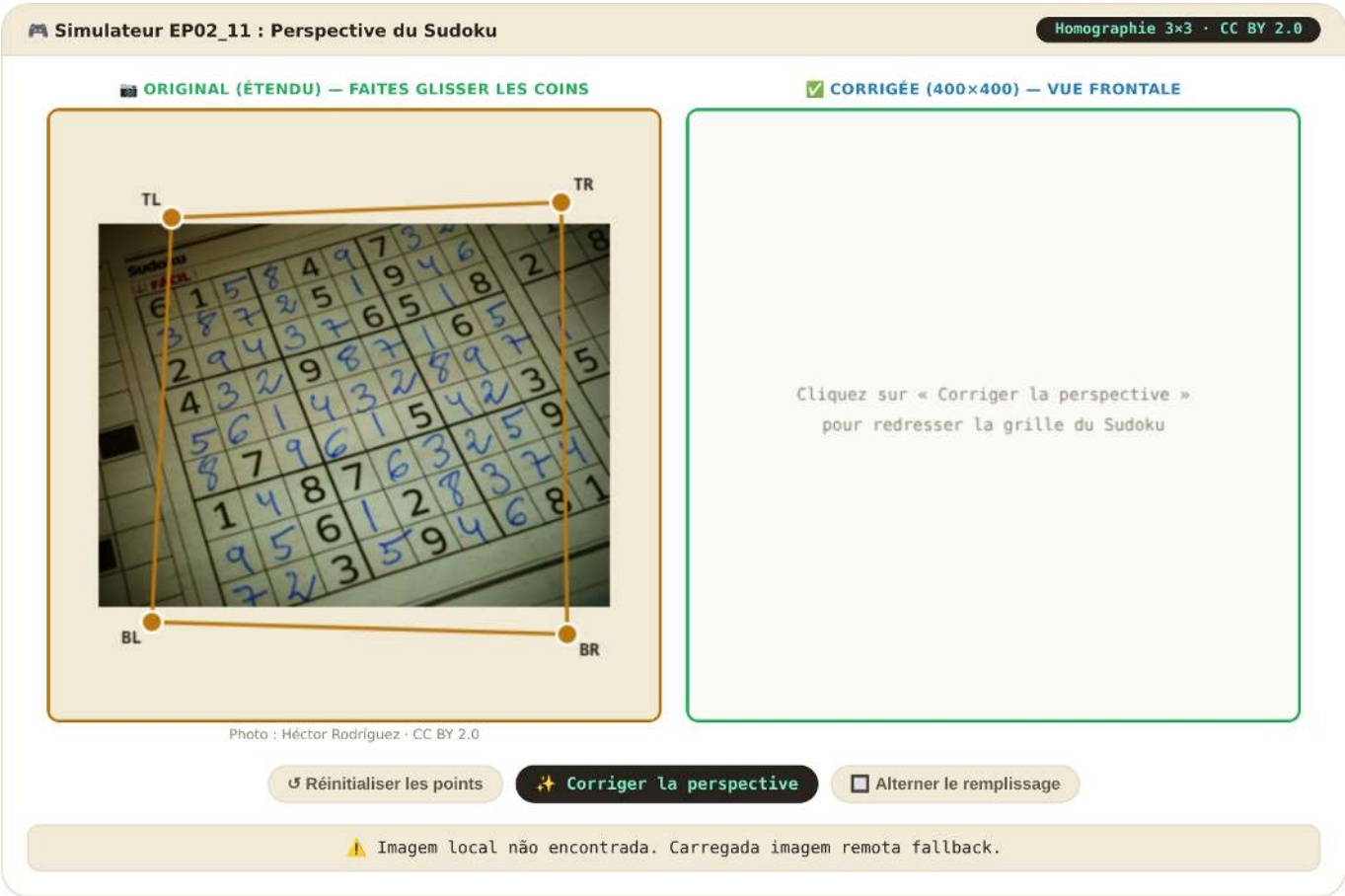


Figure 2.23: Correction de perspective : vue originale et vue frontale redressée.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap02/sim-ep0211-sudoku.html

Figure 2.24: Simulateur EP02_11 : Correction de perspective du Sudoku (Homographie 3×3 avec rééchantillonnage bilinéaire)

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

```
Overwriting EP02_11.py
```

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

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

Chapitre 3

Opérations Spatiales : Intensité, Histogramme et Filtrage

Exécuter Colab Ouvrir GitHub

Ce chapitre approfondit le traitement d'images dans le domaine spatial, en partant de la manipulation directe des pixels et des histogrammes pour le rehaussement de contraste, jusqu'à l'application de filtres locaux par convolution pour le lissage, la réduction du bruit et la détection de contours. L'objectif est de développer l'intuition mathématique et computationnelle qui sous-tend une grande partie des algorithmes modernes de Vision par Ordinateur.

3.1 Objetivos

À la fin de ce chapitre, vous serez capable de :

- **Manipuler l'intensité et les pixels :** Exécuter des opérations arithmétiques saturées (`mm.addm`, `mm.subm`) et logiques bit à bit (`mm.band`, `mm.bor`, `mm.bnot`) pour la combinaison et la sélection de régions d'intérêt (ROI), et appliquer l'*alpha blending* (`mm.blend`) pour la fusion pondérée d'images ;
- **Traiter les histogrammes :** Interpréter l'histogramme comme diagnostic des tons et appliquer l'égalisation globale via la CDF (`mm.equalize`) ; dans le parcours Python, également l'égalisation adaptative (CLAHE) et la spécification d'histogramme pour le transfert de profil tonal entre images ;
- **Comprendre les fondamentaux spatiaux :** Saisir la notion de voisinage, de *padding* des bords (`mm.pad`) et la différence entre corrélation croisée (`mm.conv`) et convolution — y compris pourquoi des *kernels* asymétriques comme celui de Sobel produisent des résultats distincts dans les deux opérations ;
- **Appliquer le filtrage de lissage :** Utiliser le filtre moyenneur (`mm.blur`, ou `mm.conv` avec un *kernel* uniforme) et le filtre gaussien (`mm.gaussian`) pour la réduction du bruit, en comprenant l'avantage de la pondération radiale et de la séparabilité gaussienne ;
- **Appliquer le filtrage de rehaussement :** Utiliser le laplacien w_4 et w_8 (`mm.laplacian`) pour le rehaussement isotrope des contours, l'opérateur de Sobel (`mm.sobel`) pour la magnitude du gradient — et, dans le parcours Python, la décomposition directionnelle G_x, G_y et l'angle —, ainsi que l'*Unsharp Masking* (`mm.usm`) pour l'amplification des hautes fréquences contrôlée par le paramètre k ;
- **Utiliser les filtres d'ordre :** Appliquer le filtre médian (`mm.median`) pour supprimer le bruit sel et poivre, en comprenant pourquoi sa nature non linéaire et sa robustesse aux *outliers* le rendent supérieur aux filtres linéaires dans ce scénario ;
- **Résoudre des problèmes pratiques :** Enchaîner les techniques dans des *pipelines* de prétraitement (égalisation → gaussien → Canny ; avec CLAHE à la place de l'égalisation dans le parcours Python) et utiliser les fonctions de `morph` (`mm.conv`, `mm.histImg`, `mm.equalize`, `mm.drawImgKernel`) pour l'analyse et la visualisation didactique de chaque étape.

3.2 Opérations au Niveau d'Intensité

Le niveau le plus élémentaire de traitement d'images agit directement sur les valeurs des pixels, sans considérer le voisinage. Ces opérations — appelées **transformations ponctuelles** (*point operations*) — sont les plus

rapides sur le plan computationnel et constituent la base de techniques plus complexes.

Formellement, une transformation ponctuelle peut être décrite comme :

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

où $f(x, y)$ est l'image d'entrée, $g(x, y)$ est la sortie et T est une fonction appliquée à chaque pixel individuellement.

3.2.1 Préparation de l'environnement pratique

Le bloc suivant charge la bibliothèque `morph` du dépôt (le module `morph.py` et, dans le parcours C++, également le `morph.hpp` utilisé dans le `#include` des cellules compilées).

```
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
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0
```

Comme objet d'étude tout au long de ce chapitre, nous utiliserons les images de vie sauvage présentées dans les Figure 3.1 et Figure 3.3.. À partir de celles-ci, nous explorerons les opérations spatiales sur l'intensité, les histogrammes et le filtrage, en analysant leurs effets sur le rehaussement, le lissage, la réduction du bruit et la détection de contours, afin de comprendre les fondements mathématiques et informatiques du TNI.

```
import os

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

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

img_color = mm.read(caminho)
img_gray  = mm.gray(img_color)

mm.show(img_color)
```

Figure 3.1: *Mandrill* (*Mandrillus sphinx*) photographié dans son environnement naturel en Afrique du Sud. Crédit : Carlos Guilherme Rodrigues (CC BY-SA 3.0).

3.2.2 Opérations Arithmétiques

Les opérations arithmétiques entre images sont largement utilisées en TDI pour combiner, comparer ou rehausser des informations. La **soustraction d'images** est particulièrement puissante pour détecter des différences entre deux trames — par exemple, lors de la suppression d'un fond statique dans les caméras de surveillance :

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

L'**addition saturée** limite le résultat à l'intervalle $[0, 255]$: les valeurs supérieures à 255 sont plafonnées à 255, évitant ainsi le *dépassement* silencieux du type `uint8` (ex. : $200 + 100 = 44$ au lieu de 300). La **soustraction saturée** applique le même principe par le bas : les valeurs négatives sont plafonnées à 0.

> ⚠ Saturation et *dépassement*
>
> Les opérations arithmétiques sur `uint8` subissent un *dépassement* silencieux : $200 + 100 = 44$ (et non 300). `mm.addm` et `mm.subm` effectuent une **saturation automatique**, en plafonnant le résultat dans $[0, 255]$. Le *mélange* utilise des poids fractionnaires : `mm.blend` opère en interne en virgule flottante, puis arrondit et sature seulement ensuite vers `uint8`.

La Figure 3.2 illustre l'addition d'une constante (éclaircissement) et la soustraction d'une constante (assombrissement avec saturation à 0).

```
fundo = 60

img_add = mm.addm(img_gray, fundo)
img_sub = mm.subm(img_gray, fundo)

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

Figure 3.2: Opérations arithmétiques saturées : addition de constante (éclaircissement) et soustraction de constante (assombrissement avec saturation à 0).

3.2.3 Pondération Pondérée (*Alpha Blending*)

La **pondération pondérée** (*alpha blending*) combine deux images en utilisant des poids complémentaires α et $(1-\alpha)$:

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

Lorsque $\alpha = 1$, seule l'image f_1 est obtenue ; lorsque $\alpha = 0$, seule f_2 l'est. Les valeurs intermédiaires produisent une transition douce entre les deux, et sont largement utilisées dans la composition d'images, la superposition de calques, les filigranes et les effets de fusion visuelle.

Pour que la combinaison produise un résultat cohérent, il est nécessaire d'aligner au préalable les régions d'intérêt. Dans la Figure 3.4, on découpe le visage du léopard avec `mm.crop(img_leop_gray, 250, H-300, 100, W-200)` et la région faciale du mandrill avec `mm.crop(img_gray, 100, 400, 380, 530)`, de sorte que les yeux et la structure faciale soient approximativement alignés. Le découpage du léopard est ensuite redimensionné (`mm.resize`) aux dimensions du mandrill avant la fusion.

`mm.blend` effectue l'opération en virgule flottante — évitant tout *dépassement* dans les calculs avec des poids fractionnaires — puis arrondit et sature le résultat en `uint8`.

```
import os

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

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

img_leop      = mm.read(caminho)
img_leop_gray = mm.gray(img_leop)

mm.show(img_leop)
```

Figure 3.3: Portrait d'un léopard (*Panthera pardus*) dans son environnement naturel. Crédit : C. Brück (CC BY-SA 4.0).

```
# découpes alignées : visage du léopard et région faciale du mandrill
leo      = mm.crop(img_leop_gray, 250, img_leop_gray.shape[0] - 300, 100,
img_leop_gray.shape[1] - 200)
mandrill = mm.crop(img_gray, 100, 400, 380, 530)
leo_r    = mm.resize(leo, (mandrill.shape[1], mandrill.shape[0]), method="bilinear")

mm.show(
    [mm.blend(mandrill, leo_r, alpha=1.0), mm.blend(mandrill, leo_r, alpha=0.8),
     mm.blend(mandrill, leo_r, alpha=0.6), mm.blend(mandrill, leo_r, alpha=0.4),
     mm.blend(mandrill, leo_r, alpha=0.2), mm.blend(mandrill, leo_r, alpha=0.0)],
    titles=[" =1.0", " =0.8", " =0.6", " =0.4", " =0.2", " =0.0"],
    cols=6
)
```

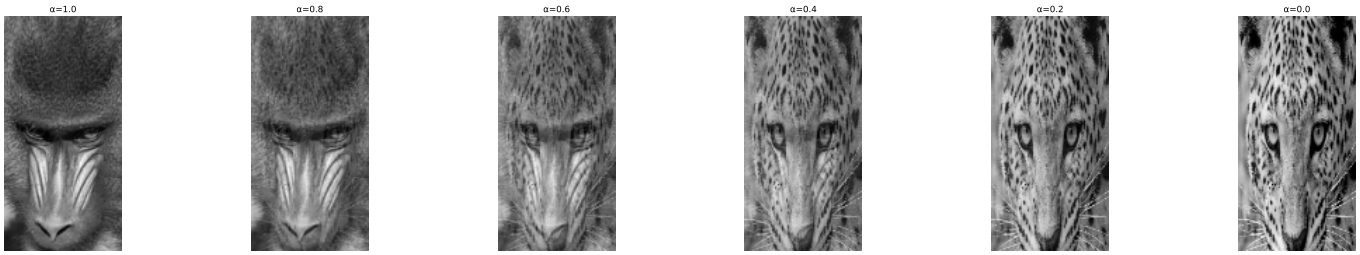


Figure 3.4: *Mélange alpha* entre des découpes alignées du mandrill et du léopard (Figure 3.3) pour différentes valeurs de . À =1, on ne voit que le mandrill ; à =0, seulement le léopard ; les valeurs intermédiaires fusionnent les regards des deux images proportionnellement.

3.2.4 Opérations Logiques et Masques Bit à Bit

Les opérations logiques bit à bit (AND, OR et NOT) agissent directement sur les bits de chaque pixel et constituent la base pour la création et l'application de **masques** (*masks*) — des images binaires avec uniquement 0 (noir) et 255 (blanc) utilisées pour isoler des **Régions d'Intérêt** (**ROI**).

Le comportement de chaque opération découle de la représentation binaire de 255 (`11111111`) et de 0 (`00000000`) :

- **AND** avec le masque : où $m = 255$, les bits d'origine sont préservés ; où $m = 0$, le pixel est mis à zéro. Résultat : découpage de la ROI.

$$g(x, y) = f(x, y) \text{ AND } m(x, y) \tag{3.4}$$

- **OR** avec le masque : où $m = 255$, le pixel est forcé au blanc ; où $m = 0$, la valeur d'origine est conservée. Résultat : éclairage de la ROI.
- **NOT** (sans masque) : inverse tous les bits ($g = 255 - f$), produisant le négatif photographique de l'image.

La Figure 3.5 illustre les trois opérations appliquées à l'image du mandrill avec un masque circulaire.

```
h, w = img_gray.shape

# Masque circulaire rempli, centré sur l'image (mm.circle dessine le
# disque ; la version didactique, test de rayon pixel par pixel, est mm.circle0).
mask_circ = mm.circle(np.zeros((h, w), dtype=np.uint8),
                      (w // 2, h // 2), min(h, w) // 3 - 10, 255, -1)

# Opérations via morph
img_not = mm.bnot(img_gray)            # NOT : négatif photographique
img_and = mm.band(img_gray, mask_circ) # préserve uniquement la ROI circulaire
img_or  = mm.bor(img_gray, mask_circ)  # illumine la région du masque

mm.show(
    [img_gray, img_and, img_or, img_not],
    titles=["Original", "AND (ROI circulaire)", "OR (illumine ROI)", "NOT (négatif)"],
    cols=4
)
```

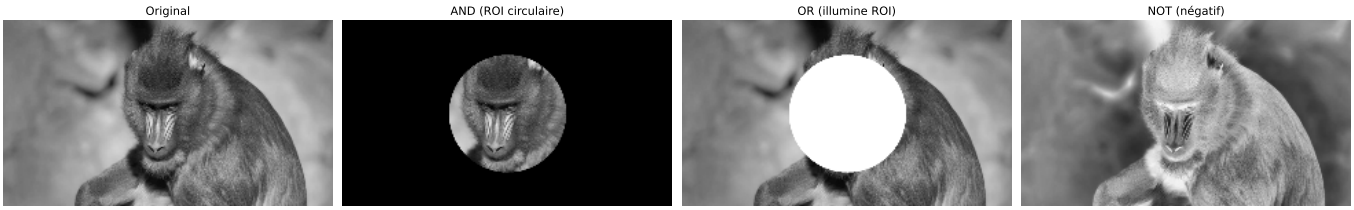


Figure 3.5: Opérations logiques bit à bit avec masque circulaire : AND (isolement de la ROI), OR (illumination de la ROI) et NOT (négatif).

3.3 Histogramme d'images

L'**histogramme** d'une image en niveaux de gris est une fonction discrète qui décrit la distribution des fréquences des intensités :

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

où r_k est le k-ième niveau d'intensité, n_k est le nombre de pixels ayant cette intensité et L est le nombre total de niveaux (typiquement 256 pour 8 bits). L'histogramme normalisé estime la probabilité de chaque niveau :

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

où MN est le nombre total de pixels. Étant une **statistique globale**, l'histogramme ne contient pas d'information positionnelle, mais révèle des caractéristiques essentielles telles que la luminosité moyenne, le

contraste et la distribution tonale. En pratique, `mm.hist(img)` retourne le vecteur des comptages $h(r_k)$, qui sert aussi bien à la visualisation (via `mm.histImg`) qu'à des calculs comme la **fonction de distribution cumulative (CDF)** et l'égalisation.

> **Interprétation de l'histogramme**
>
> - **Étroit à gauche :** image sous-exposée (sombre).
> - **Étroit à droite :** image surexposée (claire).
> - **Concentré au centre :** faible contraste.
> - **Réparti sur toute la plage :** contraste élevé, bonne utilisation des tons disponibles.

La Figure 3.6 présente l'histogramme de l'image du mandrill, ainsi que des versions assombrie (`mm.subm`) et éclaircie (`mm.addm`). On observe le déplacement de la distribution des intensités vers la gauche et vers la droite, respectivement. Notez que l'intervalle représenté sur l'axe x ne correspond pas nécessairement à toute la plage de 0 à 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=["Originale", "Assombrie (-80)", "Éclaircie (+80)",
            "Histogramme - originale", "Histogramme - assombrie", "Histogramme - éclaircie"],
    cols=3
)
```

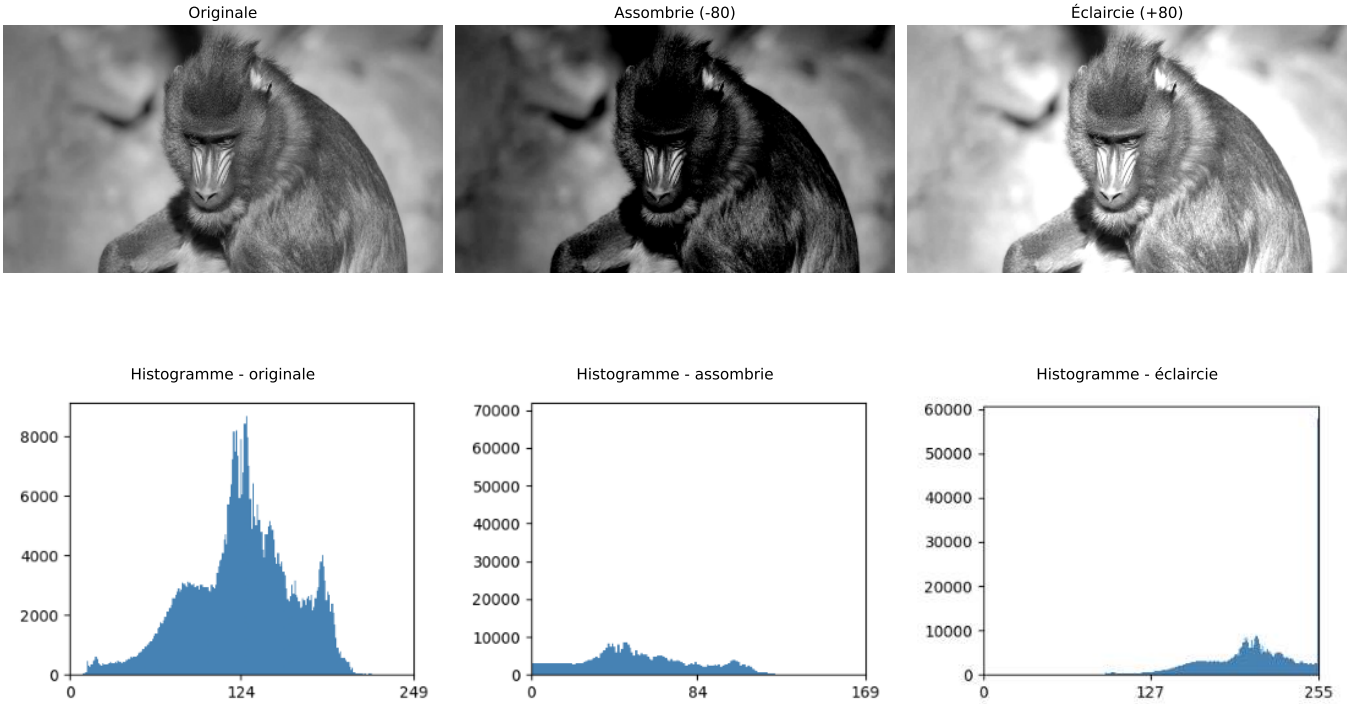


Figure 3.6: Histogrammes de l'image originale, d'une version assombrie (-80) et d'une éclaircie ($+80$). La soustraction/addition sature à 0 et 255.

3.3.1 Égalisation d'histogramme

L'**égalisation d'histogramme** redistribue les intensités afin que l'histogramme résultant soit aussi uniforme que possible. Le mappage est donné par la **fonction de distribution cumulative (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 transformation est monotone : les niveaux fréquents reçoivent des intervalles plus larges dans le domaine de sortie (plus grande séparation → plus de contraste), tandis que les niveaux rares sont compressés.

L'algorithme complet, en cinq étapes, est présenté dans la Table 3.1.

Tableau 3.1: Algorithme d'égalisation d'histogramme.

Étape	Opération	Formule
1	**Histogramme**	$h[k] \leftarrow$ nombre de pixels avec l'intensité k, $k = 0 \dots L-1$
2	**Probabilité**	$p[k] \leftarrow h[k]/MN$
3	**CDF**	$\text{cdf}[k] \leftarrow \sum_{j=0}^{k} p[j]$ (somme cumulée)
4	***Table de correspondance* (mapping)**	$\text{lut}[k] \leftarrow \text{round}(\text{cdf}[k] \times (L-1))$
5	**Application**	$g[i,j] \leftarrow \text{lut}[f[i,j]]$ (pour chaque pixel)

Notez dans la Figure 3.7 que l'égalisation **redistribue** les tons existants vers des positions plus espacées dans la plage $[0, L-1]$, mais ne crée pas de nouveaux tons — l'image égalisée conserve exactement 3 tons distincts, désormais en $\{1, 5, 7\}$ au lieu 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 : niveaux 0..7

# 1. Histogramme avec taille garantie jusqu'à L
h = mm.hist(img5, 3)  # B=3 → 2³=8 niveaux, taille garantie

p   = h / h.sum()  # 2. Probabilité de chaque niveau p[k] = h[k] / MN

cdf = np.cumsum(p) # 3. CDF normalisée ou
# cdf = np.zeros(len(p))
# cdf[0] = p[0]
# for k in range(1, len(p)):
#     cdf[k] = cdf[k-1] + p[k] # cdf[k] = Σ p[j], j=0..k

lut = np.round(cdf * (L - 1)).astype(np.uint8)  # 4. LUT : mappage vers [0, L-1]

img5_eq = lut[img5] # 5. Applique la LUT pixel par pixel ou, de manière équivalente :
# 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("Image originale (5×5, 3 bits) :")
print(img5)
print()
header = f"{'k':>3} {'h[k]':>6} {'p[k]':>7} {'CDF[k]':>8} {'lut[k]':>7}"
```

```python
print(header)
print("-" * len(header))
for k in range(L):
    print(f"{k:>3} {h[k]:>6} {p[k]:>7.4f} {cdf[k]:>8.4f} {lut[k]:>7}")
print()
print("Image égalisée (5×5) :")
print(img5_eq)

# Compte les tons distincts
tons_orig = len(np.unique(img5))
tons_eq   = len(np.unique(img5_eq))

mm.show(
    [img5, img5_eq, mm.histImg(img5, L-1), mm.histImg(img5_eq, L-1)],
    titles=[f"Original ({tons_orig} tons : {sorted(np.unique(img5).tolist())})",
            f"Égalisée ({tons_eq} tons : {sorted(np.unique(img5_eq).tolist())})",
            "Histogramme - Original",
            "Histogramme - Égalisée"],
    rows=2, cols=2, figsize=(5, 4), dpi=100
)
```

```
Image originale (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

Image égalisée (5×5) :
[[5 7 1 5 7]
 [7 5 5 7 5]
 [1 5 7 5 1]
 [5 7 5 1 5]
 [7 5 1 5 7]]
```

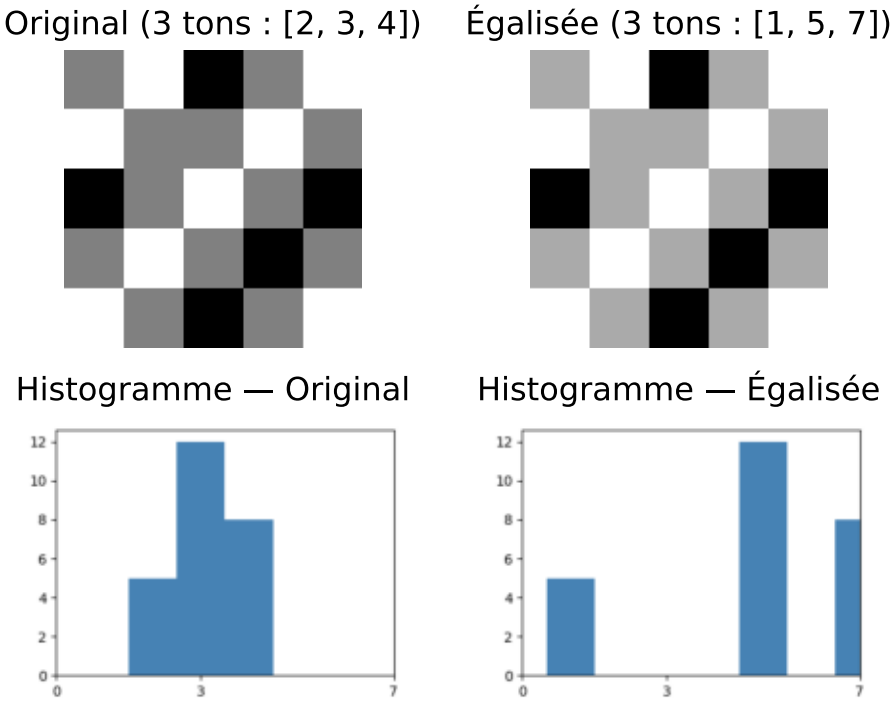


Figure 3.7: Égalisation d'histogramme sur une image 5×5 avec 3 bits (L=8) : image originale avec des tons concentrés sur {2,3,4}, image égalisée avec des tons redistribués sur {1,5,7}, et les histogrammes respectifs montrant l'étalement des fréquences.

Limitation : l'égalisation globale peut surexhausser les bruits et produire un contraste excessif dans les régions homogènes. Le **CLAHE** (*Contrast Limited Adaptive Histogram Equalization*) réduit ce problème en appliquant l'égalisation sur des blocs locaux (*tiles*) et en limitant la hauteur des pics de l'histogramme avant l'égalisation.

La Figure 3.8 compare l'image originale, l'égalisation globale via `mm.equalize` et le CLAHE d'OpenCV, en affichant également les histogrammes résultants. Contrairement à l'égalisation globale, qui utilise une transformation unique basée sur la CDF de toute l'image, le CLAHE adapte le contraste à chaque région, étant particulièrement utile pour les images avec un éclairage non uniforme.

Dans l'exemple, `clipLimit=2.0` et `tileGridSize=(32,32)` ont été utilisés. Le paramètre `clipLimit` définit dans quelle mesure les pics de l'histogramme local peuvent croître avant d'être coupés (*clipped*). Dans OpenCV, cette valeur est un facteur relatif : la limite réelle est approximativement calculée comme `clipLimit × (nombre de pixels du bloc / nombre de niveaux de gris)`. Par exemple, dans un bloc de 4096 pixels et une image 8 bits (256 niveaux de gris), la fréquence moyenne par niveau est $4096/256 = 16$. Ainsi, `clipLimit=2.0` permet des pics d'environ $2 \times 16 = 32$ occurrences avant la coupure. Les occurrences excédentaires ne sont pas rejetées : elles sont redistribuées entre les autres niveaux de gris de l'histogramme, réduisant la concentration excessive sur quelques niveaux et évitant une amplification exagérée du contraste local. Des valeurs plus faibles limitent davantage le contraste et réduisent l'amplification du bruit, tandis que des valeurs plus élevées permettent un rehaussement plus intense, mais peuvent introduire des artefacts.

- `clipLimit=1.0` : rehaussement doux et conservateur ;
- `clipLimit=2.0` : bon équilibre entre contraste et naturel ;
- `clipLimit=4.0` : mise en valeur accrue des détails locaux ;
- `clipLimit=8.0` : contraste agressif, avec amplification possible du bruit.

Ainsi, le CLAHE produit généralement des résultats plus naturels que l'égalisation globale, en particulier pour les images avec des ombres, des reflets ou un éclairage inégal.

```
img_eq    = mm.equalize(img_gray)
clahe     = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(32, 32))
img_clahe = clahe.apply(img_gray)

images = [img_gray, img_eq, img_clahe]
titles = ["Original", "mm.equalize (CDF)", "CLAHE"]

mm.show(
    images + [mm.histImg(img) for img in images],
    titles=titles + [f"Histograma - {t}" for t in titles],
    rows=2, cols=3,
    figsize=(14, 8)
)
```

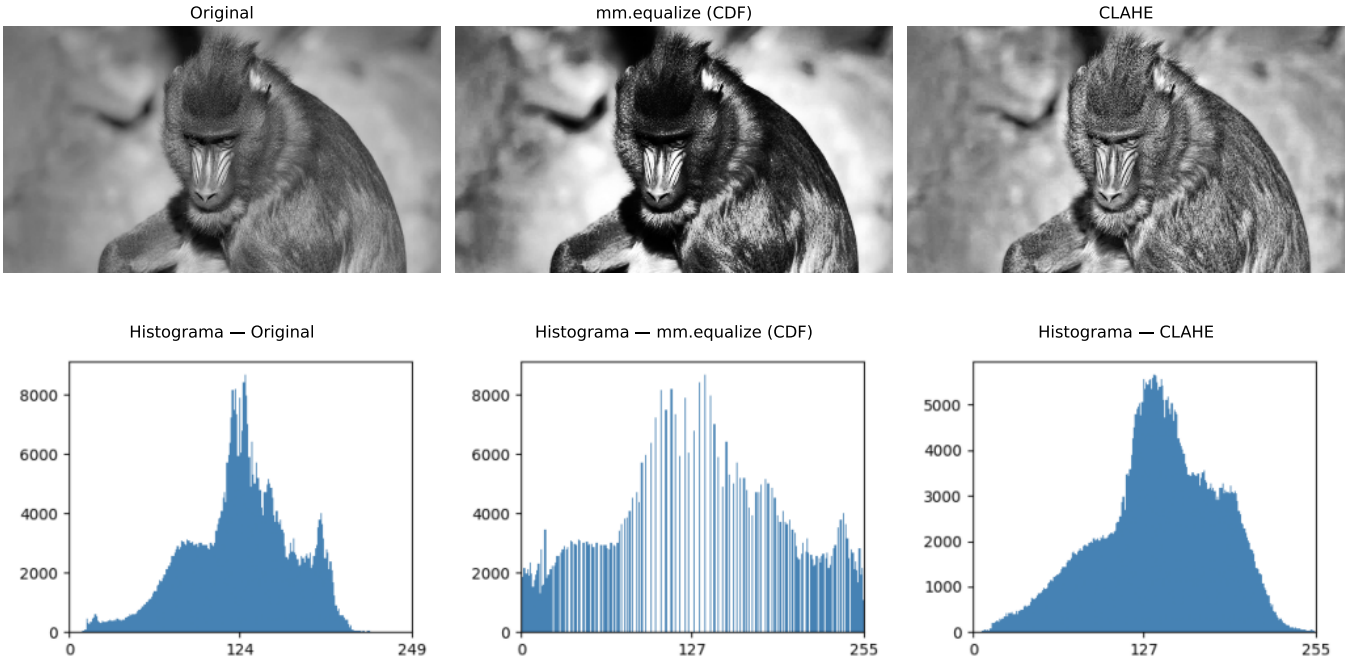


Figure 3.8: Égalisation d'histogramme : mm.equalize (global via CDF) vs. CLAHE (adaptative avec limitation de contraste). Les histogrammes révèlent l'étalement progressif des intensités.

3.3.2 Spécification d'histogramme

Alors que l'égalisation impose une distribution uniforme, la **spécification d'histogramme** (*histogram matching*) permet à l'histogramme de l'image de sortie de suivre une distribution **arbitraire** — par exemple, l'histogramme d'une autre image de référence.

La procédure comporte trois étapes :

1. Calculer la CDF de l'image d'entrée : $P_r(r_k)$.
2. Calculer la CDF de l'image de référence : $P_z(z_k)$.
3. Pour chaque niveau r_k, trouver le niveau z qui minimise $|P_z(z) - P_r(r_k)|$.

$$T(r_k) = \arg\min_z |P_z(z) - P_r(r_k)| \tag{3.8}$$

Dans Figure 3.9, nous transférons le profil tonal du léopard (Figure 3.3) vers l'image du mandrill — une application directe du concept vu dans le *blending* : au lieu de fusionner des pixels, nous fusionnons ici des distributions tonales.

```
# mappe src vers le profil tonal de ref via CDF
def hist_specify(src, ref):
    cdf_src = np.cumsum(mm.hist(src) / src.size)
    cdf_ref = np.cumsum(mm.hist(ref) / ref.size)
    lut = np.array([np.argmin(np.abs(cdf_ref - v)) for v in cdf_src], dtype=np.uint8)
    return lut[src]

img_spec = hist_specify(img_gray, img_leop_gray)

mm.show(
    [img_gray, img_leop_gray, img_spec,
     mm.histImg(img_gray), mm.histImg(img_leop_gray), mm.histImg(img_spec)],
    titles=["Mandrill (original)", "Léopard (référence)", "Mandrill → profil du léopard",
            "Histogramme - Mandrill", "Histogramme - Léopard", "Histogramme - résultat"],
    rows=2, cols=3, figsize=(12, 9)
)
```

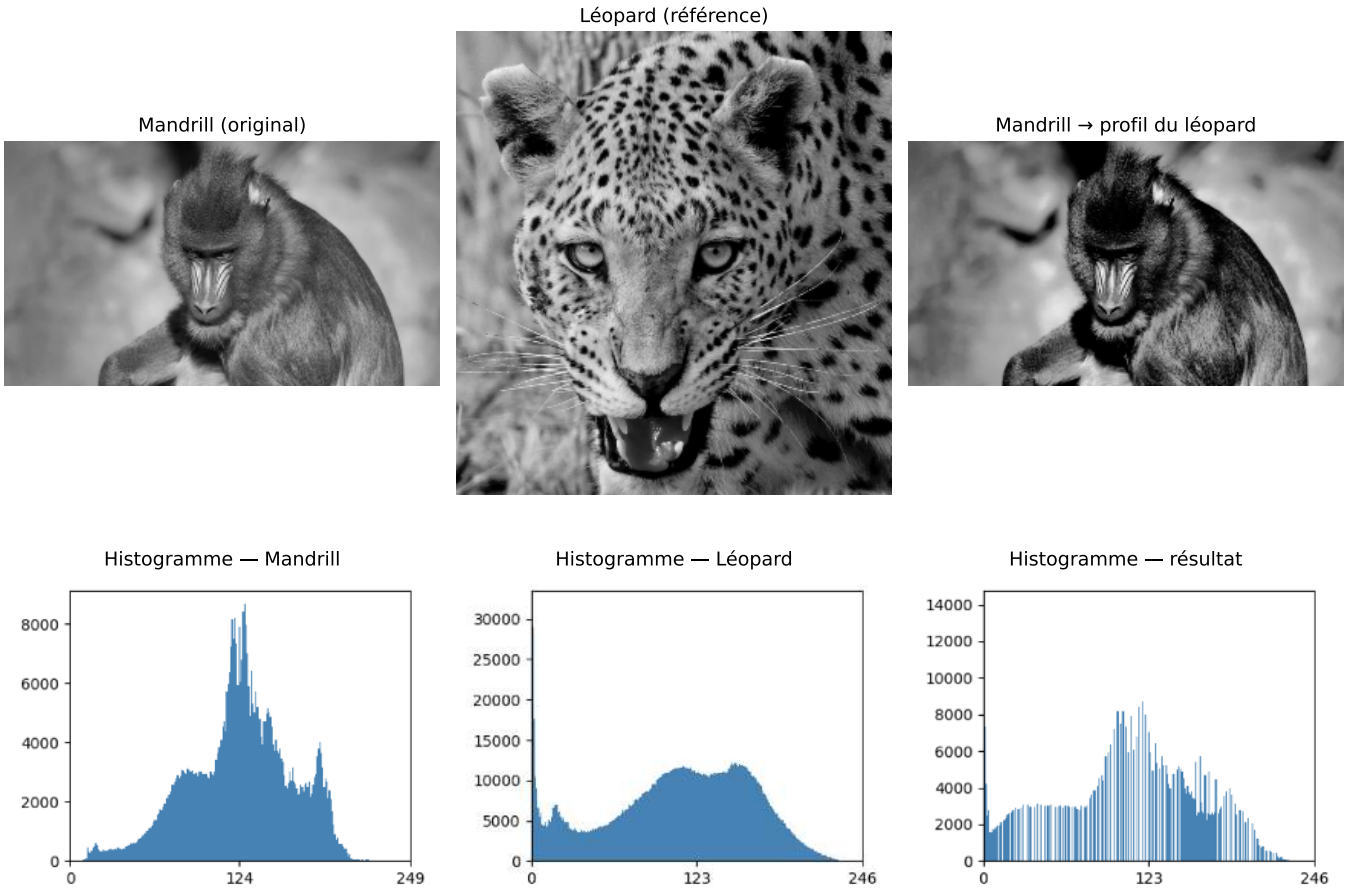


Figure 3.9: Spécification d'histogramme : mandrill mappé sur le profil tonal du léopard. La CDF de sortie approxime la CDF de référence.

3.4 Fondements spatiaux : Voisinage, convolution et *noyaux*

Les opérations de filtrage spatial n'agissent pas sur un pixel isolé, mais sur un **voisinage** qui l'entoure. Pour cela, on utilise une petite matrice de coefficients appelée **noyau** (ou masque), qui parcourt toute l'image au moyen d'une **fenêtre glissante** (*sliding window*).

Les fenêtres les plus courantes sont de taille 3×3, 5×5 et 7×7. Dans une fenêtre 3×3, par exemple, le pixel central est traité avec ses huit voisins immédiats. À chaque position de la fenêtre, les valeurs des pixels sont combinées avec les coefficients du *noyau*, produisant une nouvelle valeur pour le pixel central.

3.4.1 Voisinage

Considérons une fenêtre 3×3 centrée sur le pixel (x, y) :

$$\begin{bmatrix} (x-1, y-1) & (x, y-1) & (x+1, y-1) \\ (x-1, y) & (x, y) & (x+1, y) \\ (x-1, y+1) & (x, y+1) & (x+1, y+1) \end{bmatrix} \tag{3.9}$$

De manière générale, une fenêtre de taille $(2a+1) \times (2b+1)$ englobe tous les pixels situés jusqu'à a positions à l'horizontale et jusqu'à b positions à la verticale par rapport au pixel central. Ainsi, une fenêtre 3×3 correspond à $a = b = 1$, une fenêtre 5×5 à $a = b = 2$, et ainsi de suite.

Mathématiquement, le voisinage est défini par

$$\mathcal{V}(x, y) = \{(x+s,\, y+t) : -a \leq s \leq a,\, -b \leq t \leq b\} \tag{3.10}$$

3.4.2 Traitement des bords

Les pixels proches des bords ont une partie de leur voisinage en dehors de l'image. Pour appliquer des filtres dans ces régions, il est nécessaire de définir comment les valeurs externes seront obtenues. Les trois stratégies les plus courantes (avec la constante équivalente d'OpenCV entre parenthèses) sont :

- **Remplissage par zéros** (`BORDER_CONSTANT`) : complète la région externe avec des zéros.
- **Réplication** (`BORDER_REPLICATE`) : répète la valeur du pixel du bord.
- **Réflexion** (`BORDER_REFLECT_101`) : reflète les pixels voisins, sans répéter celui du bord.

`mm.conv` utilise la réflexion par défaut, car elle préserve mieux la continuité des niveaux de gris et réduit les artefacts dans le traitement des bords.

L'exemple suivant compare les trois stratégies avec `mm.pad` sur une matrice 3×3. Observez comment chacune remplit les pixels externes nécessaires pour appliquer un filtre 3×3 également aux coins.

```
img = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
], dtype=np.uint8)

bordas = ["constant", "replicate", "reflect101"]

print("Image originale :")
print(mm.drawImg(img))

for k in [3, 5]:
    b = k // 2
    print(f"=== Noyau {k}x{k} (rembourrage b={b}) ===")
    for nome in bordas:
        print(nome)
        print(mm.drawImg(mm.pad(img, b, border=nome)))
```

```
Image originale :
1 2 3
4 5 6
7 8 9

=== Noyau 3x3 (rembourrage 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

=== Noyau 5x5 (rembourrage 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
```

Observe que o resultado de um filtro pode variar significativamente conforme o tratamento adotado para as bordas da imagem.

Dans la bibliothèque didactique `morph.py`, les algorithmes implémentés directement en Python (sans OpenCV ni *scikit-image*) — les variantes à suffixe `0`, comme `mm.conv0` — n'utilisent pas de *padding*. Pour chaque pixel, les éléments du voisinage sont examinés individuellement, et seuls les voisins dont les coordonnées appartiennent au domaine de l'image participent au calcul. Ainsi, dans les régions proches des bords, le voisinage effectif peut contenir moins de pixels que la fenêtre spécifiée ; dans les opérateurs basés sur `mm._viz` (par exemple `mm.laplacian`), les pixels de bord avec une fenêtre incomplète sont laissés inchangés.

3.4.3 Corrélation vs. Convolution

Il existe deux mécanismes mathématiquement liés.

Corrélation croisée (*cross-correlation*) — le *noyau* est appliqué directement :

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)\, f(x+s,\, y+t) \tag{3.11}$$

Convolution bidimensionnelle — le *noyau* est pivoté de 180° avant son application :

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)\, f(x-s,\, y-t) \tag{3.12}$$

Pour les *noyaux* symétriques (gaussien, laplacien, moyenne), les deux opérations produisent des résultats identiques. Pour les *noyaux* asymétriques (Sobel, Prewitt), la différence est significative, comme le montrent les exemples suivants.

3.4.3.1 Corrélation (`mm.conv`)

```
# image 4x4 (uint8) et noyau asymétrique 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")  # zéro hors de l'image

print("Image originale :");         print(mm.drawImg(img))
print("Noyau :");                  print(mm.drawImg(w))
print("Résultat de la corrélation :"); print(mm.drawImg(corr))
```

```
Image originale :
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16

Noyau :
  0   1   2
  0   0   0
  0   0   0

Résultat de la corrélation :
 0  0  0  0
 5  8 11  4
17 20 23  8
29 32 35 12
```

`mm.conv` effectue une corrélation, c'est-à-dire qu'il applique le *noyau* exactement dans l'orientation fournie.

3.4.3.2 Convolution

```
# répétés ici pour que la cellule soit indépendante
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)

# noyau tourné de 180° (équivaut à 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("Image originale :");            print(mm.drawImg(img))
print("Noyau original :");            print(mm.drawImg(w))
print("Noyau tourné de 180° :");    print(mm.drawImg(w_conv))
print("Résultat de la convolution :");    print(mm.drawImg(conv))
```

```
Image originale :
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16

Noyau original :
  0   1   2
  0   0   0
  0   0   0

Noyau tourné de 180° :
  0   0   0
  0   0   0
  2   1   0

Résultat de la convolution :
 5 16 19 22
 9 28 31 34
13 40 43 46
 0  0  0  0
```

La convolution utilise le *noyau* tourné de 180°. Pour reproduire la définition mathématique de la convolution, on tourne le *noyau* (ici, `[[0,1,2],[0,0,0],[0,0,0]]` → `[[0,0,0],[0,0,0],[2,1,0]]`) avant d'appliquer `mm.conv`.

3.4.4 Le Rôle du *Noyau*

Les coefficients du *noyau* déterminent complètement l'effet produit par le filtre, comme résumé dans la Table 3.2.

Tableau 3.2: Interprétation typique des coefficients du *noyau*.

Caractéristique	Effet typique
Coefficients positifs dont la somme vaut 1	Lissage (*passe-bas*)
Somme égale à 0, avec des valeurs positives et négatives	Détection de contours (*passe-haut*)
Coefficient central positif dominant et voisins négatifs	Renforcement de la netteté
Coefficients asymétriques	Gradient directionnel

Exemples :

Lissage :

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

Détection de contours :

$$\begin{bmatrix}-1 & -1 & -1\\ -1 & 8 & -1\\ -1 & -1 & -1\end{bmatrix}$$

Renforcement de la netteté :

$$\begin{bmatrix}0 & -1 & 0\\ -1 & 5 & -1\\ 0 & -1 & 0\end{bmatrix}$$

Gradient directionnel (Sobel) :

$$\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix}$$

La Figure 3.10 démontre le mécanisme étape par étape : pour chaque position de la fenêtre, on multiplie chaque coefficient du *kernel* par le pixel correspondant de la voisinage et on additionne les produits obtenus. Le résultat est exactement la valeur définie par la Équation 3.11 pour cette position de l'image. Bien que les images produites par `mm.conv0` et `cv2.filter2D` (ou `mm.conv`) soient visuellement très similaires, l'implémentation basée sur OpenCV est des milliers de fois plus rapide, comme montré ci-dessous.

Performance : boucles Python vs. opérations vectorisées

La fonction `mm.conv0` implémente la corrélation directement en Python au moyen de boucles imbriquées. Bien que cette approche soit adaptée à des fins pédagogiques, elle exécute un grand nombre d'opérations et devient lente pour des images de grande taille.
Quant à `mm.conv`, elle utilise `cv2.filter2D`, implémenté en C++ et optimisé pour les opérations matricielles. Dans l'exemple présenté, la version vectorisée a été plus de **3000 fois plus rapide** que l'implémentation pédagogique, produisant un résultat visuellement équivalent.
Les différences numériques observées se concentrent principalement sur les bords de l'image. Dans `mm.conv0`, les pixels des bords restent inchangés, tandis que `mm.conv` utilise une stratégie de réflexion des bords (`cv2.BORDER_REFLECT_101`, valeur par défaut de `cv2.filter2D`).
C'est pourquoi `mm.conv0` doit être utilisée pour comprendre l'algorithme, tandis que `mm.conv` est l'option recommandée pour les applications pratiques.

```
import time

w_mean = np.ones((3,3), dtype=np.float32) / 9.0
img_gray = img_leop_gray
# Démonstration numérique sur un patch 5×5
patch = img_gray[250:255, 250:255].astype(np.float32)
roi   = patch[0:3, 0:3]
print("Patch 5×5 (intensités) :")
print(patch.astype(np.int32))
print(f"\nNoyau 3×3 de moyenne :\n{w_mean}")
print(f"\nCorrélation sur le pixel central [1,1] : \
      {(w_mean * roi).sum():.1f}  (original : {patch[1,1]:.0f})")

# Comparaison de temps sur l'image complète
t0 = time.perf_counter()
img_conv0 = mm.conv0(img_gray, w_mean)
t_conv0 = time.perf_counter() - t0
```

```
t0 = time.perf_counter()
img_conv = mm.conv(img_gray, w_mean)
t_conv = time.perf_counter() - t0

diff = np.abs(img_conv0.astype(int) - img_conv.astype(int)).max()
print(f"\nTemps sur l'image {img_gray.shape[0]}x{img_gray.shape[1]} :")
print(f"  mm.conv0 (boucles Python) : {t_conv0:.3f} s")
print(f"  mm.conv  (cv2.filter2D) : {t_conv:.5f} s")
print(f"  Accélération :             {t_conv0/t_conv:.0f}× plus rapide")
print(f"  Différence maximale :       {diff} (bords avec padding différent)")

mm.show(
    [img_gray, img_conv0, img_conv],
    titles=["Original", f"conv0 ({t_conv0:.2f}s)", f"conv ({t_conv:.4f}s)"],
    cols=3
)
```

```
Patch 5×5 (intensités) :
[[ 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]]

Noyau 3×3 de moyenne :
[[0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]]

Corrélation sur le pixel central [1,1] :       103.0  (original : 107)

Temps sur l'image 1324x1242 :
  mm.conv0 (boucles Python) : 6.010 s
  mm.conv  (cv2.filter2D) : 0.00270 s
  Accélération :             2228× plus rapide
  Différence maximale :       39 (bords avec padding différent)
```

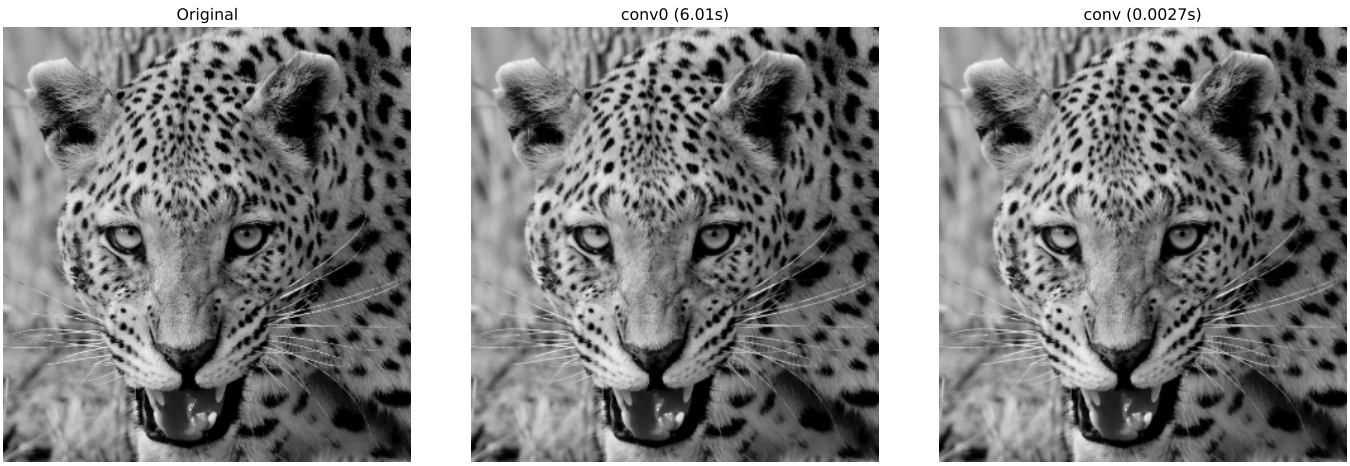


Figure 3.10: Corrélation avec *noyau* de moyenne 3×3 : version didactique (mm.conv0) vs. vectorisée (mm.conv via cv2.filter2D). La différence maximale de 39 se produit sur les bords.

3.4.5 Exemple Numérique : Corrélation Pas à Pas

Pour rendre concret le mécanisme de la Équation 3.11, considérez le *noyau* de moyenne 3×3 ($a = b = 1$, tous les coefficients $= 1/9 \approx 0{,}111$) appliqué au *patch* 5×5 extrait de l'image du léopard. La Figure 3.11 affiche le *patch* avec une grille et met en évidence en jaune la fenêtre 3×3 centrée sur le pixel $[1, 1]$:

```
patch = img_gray[250:255, 250:255]
B     = np.ones((3, 3), dtype=np.uint8)

print("Patch 5×5 (intensités) :")
print(mm.drawImg(patch))

mm.drawImgKernel(patch, B, 1, 1, 40)
```

```
Patch 5×5 (intensités) :
 91  95 108 116 114
106 107 108 111 114
102 103 107 112 116
 83  90 111 125 123
 79  88 110 126 124
```

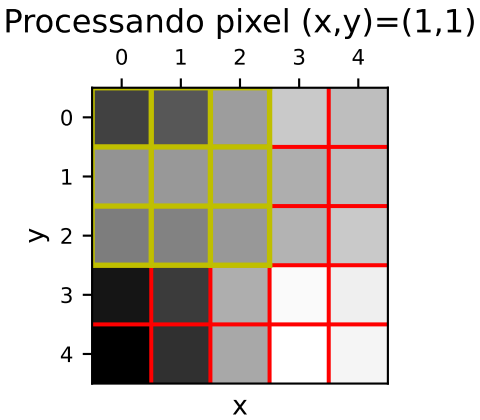


Figure 3.11: *Patch* 5×5 extraído da imagem do leopardo (posição [250:255, 250:255]). A janela amarela destaca a vizinhança 3×3 centrada no pixel [1,1] onde a correlação será calculada.

Pour illustrer le calcul de la corrélation, considérons le pixel à la position $[1, 1]$ du *patch* 5×5 montré dans la Figure 3.11.. Cette position a été choisie uniquement pour des raisons pédagogiques, car elle possède un voisinage 3×3 complet autour d'elle.

Les valeurs de ce voisinage correspondent à la sous-matrice supérieure gauche du *patch* :

$$\text{vizinhança} = \begin{bmatrix} 91 & 95 & 108 \\ 106 & 107 & 108 \\ 102 & 103 & 107 \end{bmatrix}$$

En appliquant la Équation 3.11 avec le noyau de moyenne :

$$g[1, 1] = \frac{91 + 95 + 108 + 106 + 107 + 108 + 102 + 103 + 107}{9} = \frac{927}{9} = 103$$

Le résultat (103) est légèrement inférieur à la valeur originale du pixel central (107), car la moyenne intègre des voisins de moindre intensité, produisant l'effet de lissage. En pratique, l'algorithme démarre le traitement

à $[0, 0]$ et répète ce même calcul pour chaque position de l'image, déplaçant la fenêtre jusqu'à couvrir tout le domaine.

3.5 Filtrage Spatial de Lissage

Les filtres de lissage (*smoothing filters*) atténuent les variations brusques d'intensité, réduisant le bruit et les détails à haute fréquence. Ce sont des filtres **passe-bas** — ils préservent les composantes de basse fréquence (structures de grande taille) et atténuent celles de haute fréquence (bruit, contours).

3.5.1 Filtre Moyenneur (*Box Filter*)

Le filtre moyenneur utilise un *noyau* uniforme de taille $n \times n$, dont tous les coefficients valent $1/n^2$:

$$w_{\text{moyenne}} = \frac{1}{n^2} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}_{n \times n} \tag{3.13}$$

Chaque pixel de sortie est la moyenne arithmétique des n^2 pixels de son voisinage. Notez que la somme des coefficients est toujours égale à 1 — la luminosité moyenne de l'image est préservée. Des *noyaux* plus grands produisent un lissage plus agressif, mais estompent progressivement les contours.

La Figure 3.12 montre l'effet du filtre de moyenne avec des *noyaux* 3×3, 7×7 et 15×15 sur un détail de l'image du léopard. Les résultats ont été obtenus avec `mm.blur`, qui implémente le filtre de moyenne via la fonction `cv2.blur`, équivalente à la convolution de l'image avec un *noyau* uniforme dont les coefficients sont $h(x, y) = 1/N^2$; de manière équivalente, le même résultat peut être obtenu avec `mm.conv`, en calculant $(g = f * h)$. À mesure que le *noyau* augmente, davantage de pixels contribuent à chaque valeur de sortie, intensifiant le lissage, réduisant le bruit et rendant les détails fins et les bords progressivement plus flous.

```
# Détail de la région de l'œil
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] # ou
    #[mm.conv(img_gray_crop, np.ones((k,k), dtype=np.float32)/(k*k)) for k in sizes]
titles = ["Original"] + [f"Média {k}×{k}" for k in sizes]

mm.show(imgs, titles=titles, cols=4)
```

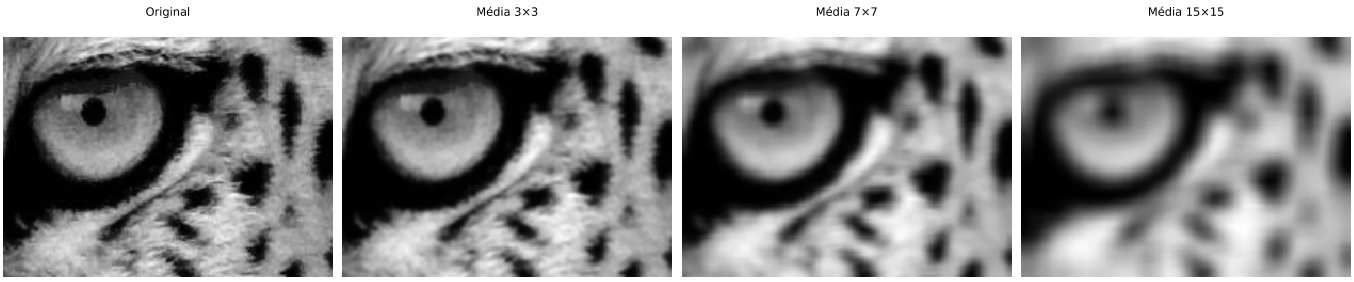


Figure 3.12: Filtre de moyenne avec des *kernels* de taille croissante (3×3, 7×7, 15×15). Le flou des bords augmente avec la taille du *kernel*.

3.5.2 Filtre gaussien

Le filtre gaussien pondère les pixels du voisinage selon une fonction gaussienne bidimensionnelle :

$$G(s,t) = \frac{1}{2\pi\sigma^2}\, e^{-\frac{s^2+t^2}{2\sigma^2}} \tag{3.14}$$

où σ est l'écart type et contrôle le rayon d'influence. Les pixels plus proches du centre ont un poids plus important ; les pixels éloignés sont progressivement ignorés.

La Figure 3.13 présente le *noyau* gaussien 5×5 généré pour $\sigma = 1$. Le *noyau* a été construit à partir du produit externe de deux vecteurs gaussiens unidimensionnels, puis normalisé pour que la somme de ses coefficients soit égale à 1. On observe que les poids les plus importants se concentrent au centre de la matrice, en décroissant radialement vers les bords. Cette distribution fait que les pixels centraux ont une plus grande influence sur le résultat du filtrage, contribuant à un lissage plus naturel et à une meilleure préservation des contours que le filtre de moyenne.

```
# Génère le noyau gaussien 5×5 via cv2
k_gauss = cv2.getGaussianKernel(5, 1)
w_gauss = k_gauss @ k_gauss.T          # @ => multiplication matricielle : G(s,t) = G(s)·G(t)
w_gauss = w_gauss / w_gauss.sum()      # garantit somme = 1

print("Noyau gaussien 5×5 ( =1), normalisé :")
for row in w_gauss:
    print("  " + "  ".join(f"{v:.4f}" for v in row))
print(f"\nSomme des coefficients : {w_gauss.sum():.6f}")
print(f"Poids central vs. coin : {w_gauss[2,2]:.4f} vs. {w_gauss[0,0]:.4f} "
      f"({w_gauss[2,2]/w_gauss[0,0]:.1f}× plus grand]")

mm.drawImgPlt((w_gauss * 1000).astype(np.uint8), scale=40)
```

```
Noyau gaussien 5×5 ( =1), normalisé :
  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

Somme des coefficients : 1.000000
Poids central vs. coin : 0.1621 vs. 0.0030 (54.6× plus grand]
```

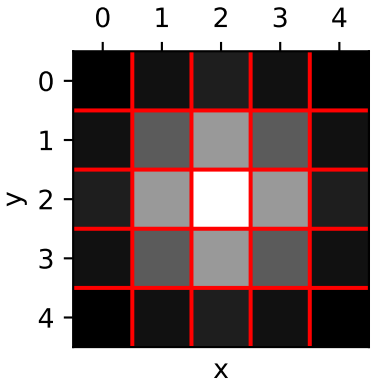


Figure 3.13: *Noyau* gaussien 5×5 (=1) : poids normalisés, plus grands au centre et décroissant radialement. La grille facilite la lecture de chaque coefficient.

Avantage computationnelle de la séparabilité

Considérons un *noyau* carré de taille $n \times n$. Si ce filtre est séparable (comme le filtre gaussien), la convolution 2D peut être décomposée en deux convolutions 1D : une horizontale et une verticale. Dans ce cas, le coût par pixel passe d'environ $O(n^2)$ opérations (convolution 2D directe) à $O(2n)$ opérations (deux convolutions 1D). Ainsi, la complexité est réduite de manière significative, rendant le traitement plus efficace.

Comparé au filtre moyenneur, le filtre gaussien :

- **Préserve mieux les contours** — la pondération radiale lisse sans créer de transitions abruptes ;
- **N'introduit pas d'effet d'anneau** (*ringing*) dans le domaine fréquentiel, car la gaussienne est sa propre transformée de Fourier (Chapitre 5) ;
- **Est contrôlé par** σ — augmenter σ équivaut à augmenter le rayon de lissage de manière continue et prévisible.

La Figure 3.14 compare les filtres de moyenne et gaussien appliqués à l'image du léopard à l'aide d'une fenêtre 9×9. Le filtre de moyenne a été implémenté par convolution avec un *noyau* uniforme, où tous les 81 pixels du voisinage possèdent le même poids (1/81), tandis que le filtre gaussien a été obtenu avec `cv2.GaussianBlur`, utilisant des poids définis par une distribution gaussienne. Tous deux réduisent le bruit et lissent l'image, mais le filtre gaussien préserve mieux les contours et les détails locaux, comme on peut l'observer dans la région agrandie de l'œil.

La Figure 3.14 compare les filtres de moyenne et gaussien appliqués à un détail de l'image du léopard avec des *noyaux* 9×9. Le filtre de moyenne a été obtenu avec `mm.blur`, équivalent à la convolution avec un *noyau* uniforme dont les coefficients valent 1/81, tandis que le filtre gaussien a été obtenu avec `mm.gaussian`, équivalent à la convolution avec un *noyau* généré à partir d'une distribution gaussienne. Tous deux favorisent le lissage et la réduction du bruit, mais le filtre gaussien attribue un poids plus important aux pixels centraux du voisinage, préservant mieux les contours et les détails locaux, comme on peut l'observer dans la région agrandie de l'œil.

```
img_media9 = mm.blur(img_gray, 9)
img_gauss9 = mm.gaussian(img_gray, 9, 0)

# Détail de la région de l'œil
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=["Détail : Original", "Moyenne 9×9", "Gaussien 9×9"],
    cols=3, figsize=(12, 8)
)
```

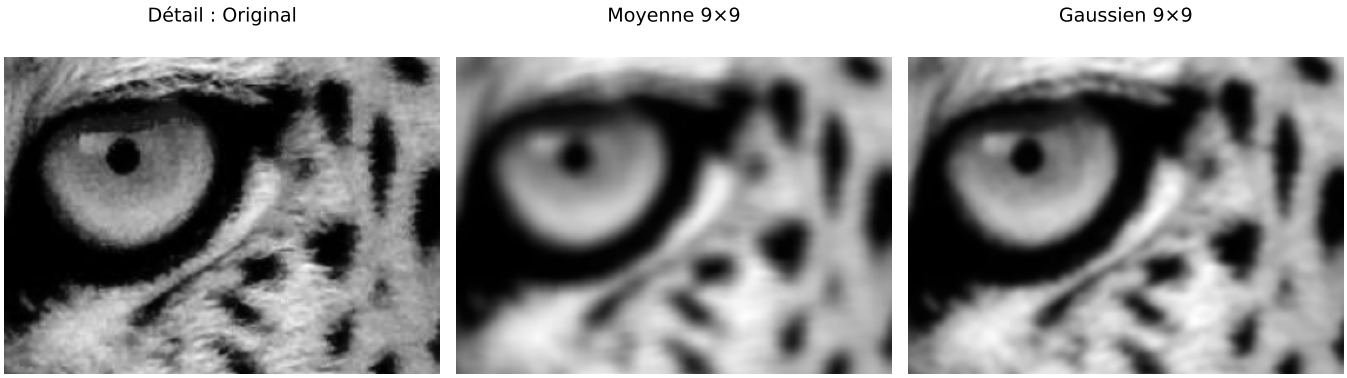


Figure 3.14: Comparaison entre filtre de moyenne et gaussien (*noyau* 9×9, =0). Le gaussien préserve mieux les bords, visible dans le détail du visage.

3.6 Filtrage Spatial de Rehaussement

Les filtres de rehaussement (*sharpening filters*) mettent en évidence les transitions abruptes d'intensité, augmentant la netteté et la visibilité des contours. Ce sont des filtres **passe-haut** — ils amplifient les composantes haute fréquence (contours, texture) et suppriment celles de basse fréquence (régions uniformes).

L'intuition est simple : si l'on soustrait d'une image sa version lissée (qui ne contient que les basses fréquences), ce qui reste correspond aux hautes fréquences — contours et détails. En ajoutant ce résidu à l'image originale, le contraste local augmente :

$$g = f + k\,(f - f_{\text{lisse}}), \quad k > 0 \tag{3.15}$$

La Figure 3.15 illustre ce processus sur un signal 1D synthétique avec trois structures distinctes : un échelon large, un pic fin et une rampe douce. Dans le panneau , le signal original $f(x)$; dans le , la version lissée $f_{\text{lisse}}(x)$ obtenue par moyenne mobile — on note comment le pic fin est atténué. Le panneau affiche le résidu $f - f_{\text{lisse}}$, qui ne conserve que les transitions abruptes. Enfin, le panneau montre $g(x)$: le pic, auparavant atténué, est restauré et amplifié par rapport à l'original. Ajustez k et la taille de la fenêtre pour observer le *trade-off* entre netteté et amplification du bruit.

Les filtres de rehaussement formalisent cette idée directement dans le *kernel*, sans avoir besoin de deux étapes séparées.

3.6.1 Laplacien

Le Laplacien est un opérateur isotrope de **dérivée seconde**, c'est-à-dire qu'il répond de manière identique aux variations dans toutes les directions, contrairement aux opérateurs de dérivée première, comme Sobel et Prewitt, qui sont directionnels :

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{3.16}$$

Une propriété importante de la dérivée seconde est que sa valeur est **proche de zéro dans les régions uniformes** et élevée aux transitions d'intensité. Ainsi, en soustrayant le Laplacien de l'image originale, on renforce les contours et les détails, augmentant le contraste local :

$$g(x, y) = f(x, y) - \nabla^2 f(x, y) \tag{3.17}$$

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap03/sim-03-filtragem1d.html

Figure 3.15: Simulateur : Filtrage spatial d'accentuation 1D (Unsharp Masking et High-Boost)

Sous forme discrète, la dérivée seconde en x est approchée par $f(x+1,y) - 2f(x,y) + f(x-1,y)$, et de manière analogue en y. En sommant les deux directions, on obtient le *kernel* w_4 (4-voisins) ou w_8 (8-voisins, incluant les diagonales) :

$$w_4 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \qquad w_8 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.18}$$

Somme nulle et centre négatif

Les deux *kernels* ont une **somme des coefficients égale à zéro** : dans les régions uniformes, la sortie est 0 — le Laplacien ne modifie pas la luminosité moyenne, il détecte uniquement les variations. Le **centre négatif** indique que le pixel est comparé à ses voisins : plus il se démarque (vers le haut ou vers le bas), plus la valeur absolue du Laplacien en ce point est grande.

Dans l'exemple suivant, le pixel central $[1,1] = 107$ possède des voisins $\{95, 106, 108, 103\}$. Comme ces valeurs sont proches les unes des autres, la région est presque uniforme et le Laplacien retourne une valeur faible, produisant peu de rehaussement. Dans les régions de bord, où il y a des différences plus grandes entre le pixel central et ses voisins, le Laplacien prend des valeurs plus élevées (positives ou négatives), et l'opération de Équation 3.17 intensifie ces transitions.

La Figure 3.16 illustre le calcul du Laplacien avec le *noyau* w_4 dans un voisinage 3×3 mis en évidence à l'intérieur d'un *patch* 5×5. L'exemple montre la valeur obtenue par l'opérateur et le pixel correspondant rehaussé dans l'image de sortie, qui passe de 107 à 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("Patch 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"Pixel rehaussé [1,1] : {p[1,1]:.0f} - ({lap_11}) = {real_11}")

mm.drawImgKernel(patch, B, x=1, y=1, scale=40.0)
```

```
Patch 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
Pixel rehaussé [1,1] : 107 - (-16) = 123
```

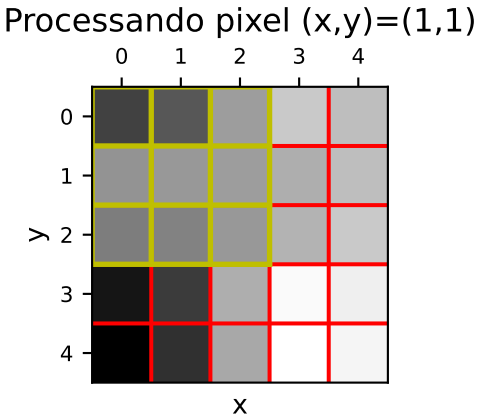


Figure 3.16: *Noyau* Laplacien w4 appliqué au *patch* 5×5 : la fenêtre jaune met en évidence le voisinage 3×3 où l'opérateur de dérivée seconde est calculé.

La Figure 3.17 compare l'application des *kernels* Laplaciens w_4 et w_8 sur un recadrage plus grand de l'image du léopard. Pour chaque cas, sont présentées la réponse brute de l'opérateur, qui met en évidence les contours et les transitions d'intensité, ainsi que l'image obtenue après la rehaussement par soustraction du Laplacien. On observe que le *kernel* w_8, en considérant également les voisins diagonaux, produit une réponse plus intense et détecte des variations dans davantage de directions, résultant en un rehaussement légèrement plus accentué.

```
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", "Laplacien w4", "Rehaussement w4",
            "Original", "Laplacien w8", "Rehaussement w8"],
    cols=3, figsize=(14, 8)
)
```

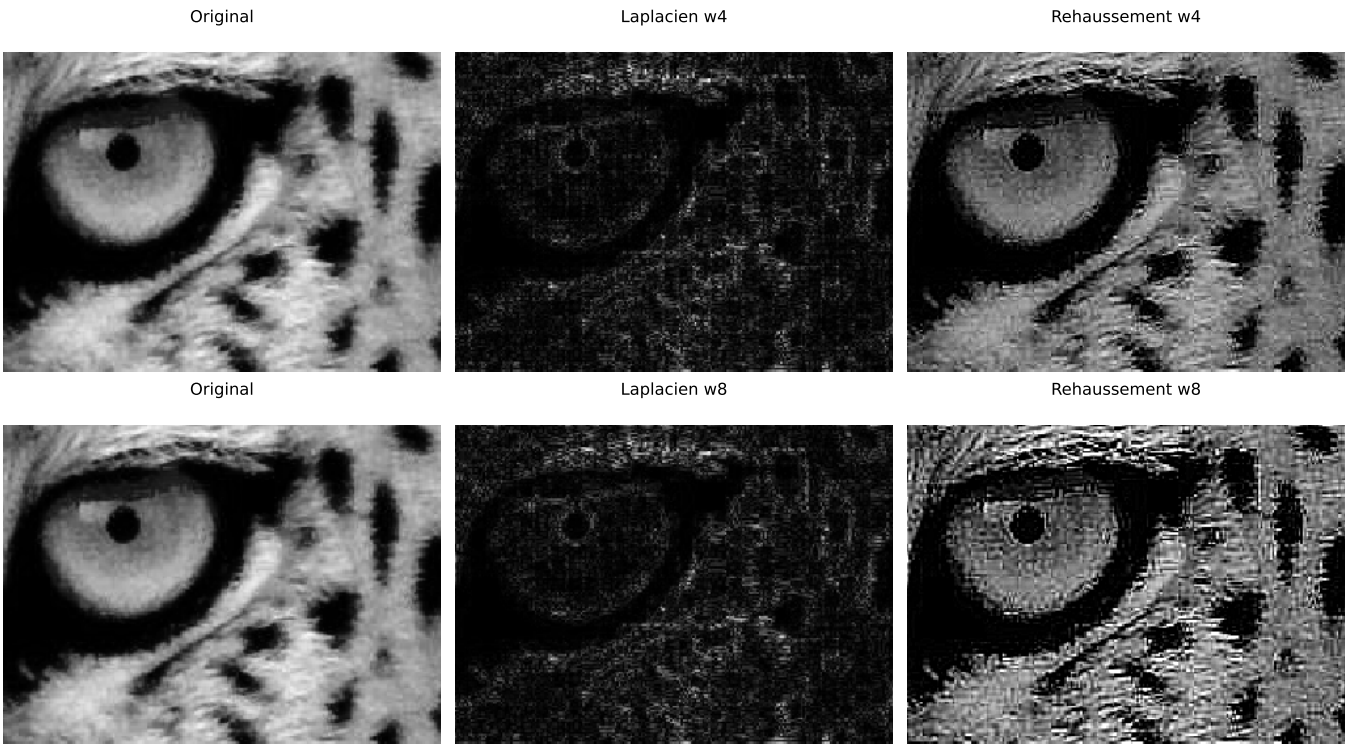


Figure 3.17: Laplacien appliqué à l'image du léopard : réponse brute (bords) avec w4 et w8, et images rehaussées par la soustraction du Laplacien. w8 est plus sensible aux diagonales.

3.6.2 Opérateur de Sobel

L'opérateur de Sobel estime les **dérivées partielles du premier ordre** dans les directions horizontale et verticale. Contrairement au Laplacien (dérivée seconde), Sobel est directionnel et plus robuste au bruit, car chaque *kernel* combine une dérivée avec un lissage gaussien perpendiculaire :

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * f, \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * f \tag{3.19}$$

G_x détecte les bords **verticaux** (variation dans la direction x) ; G_y détecte les bords **horizontaux** (variation dans la direction y). Les poids $\{1, 2, 1\}$ dans la direction perpendiculaire correspondent au lissage gaussien 1D, qui réduit la sensibilité au bruit.

> **ℹ Sobel est une corrélation, pas une convolution**
>
> Les *kernels* de Sobel sont **asymétriques** — la rotation de 180° modifie le résultat. `cv2.Sobel` implémente une corrélation croisée (comme `cv2.filter2D`). Pour obtenir la dérivée directionnelle correcte, les signes sont déjà définis pour la corrélation : G_x retourne des valeurs positives lorsque l'intensité croît de la gauche vers la droite.

La magnitude du **gradient** combine les deux composantes, représentant la force du bord indépendamment de la direction :

$$|\nabla f| = \sqrt{G_x^2 + G_y^2} \tag{3.20}$$

Et la **direction** du gradient (perpendiculaire au bord) est :

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \tag{3.21}$$

Pour illustrer numériquement, on calcule G_x et G_y manuellement au pixel central $[1,1]$ du *patch* 5×5 :

```
patch = img_gray[250:255, 250:255]
p     = patch.astype(np.float32)

# Gx au pixel [1,1] : colonne droite - colonne gauche (pondérée)
gx = (p[0,2] + 2*p[1,2] + p[2,2]) - (p[0,0] + 2*p[1,0] + p[2,0])

# Gy au pixel [1,1] : ligne inférieure - ligne supérieure (pondérée)
gy = (p[2,0] + 2*p[2,1] + p[2,2]) - (p[0,0] + 2*p[0,1] + p[0,2])

mag   = np.sqrt(gx**2 + gy**2)
theta = np.degrees(np.arctan2(gy, gx))

print("Patch 5x5 :")
print(mm.drawImg(patch))
print(f"Gx 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}")

# Remplacé le | par le préfixe mag pour ne pas casser l'interpréteur de tableaux de 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} degrés")
```

```
Patch 5x5 :
 91  95 108 116 114
106 107 108 111 114
102 103 107 112 116
 83  90 111 125 123
 79  88 110 126 124

Gx 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 degrés
```

La valeur réduite de $|\nabla f|$ dans ce *patch* confirme que la région est presque uniforme, car le gradient prend des valeurs élevées uniquement là où il y a des changements significatifs d'intensité. La Figure 3.18 applique l'opérateur de Sobel à un recadrage plus large de l'image du léopard. Sont présentées les réponses horizontale (G_x) et verticale (G_y), obtenues par convolution avec les *kernels* de Sobel respectifs, ainsi que la magnitude $|\nabla f|$, calculée à partir de la combinaison des deux. Alors que G_x met en évidence les bords verticaux et G_y les bords horizontaux, la magnitude souligne les bords dans toutes les directions.

```
def norm255(img):
    mn, mx = img.min(), img.max() #+1e-9 pour éviter la division par zéro
    return ((img - mn) / (mx - mn + 1e-9) * 255).astype(np.uint8)

sobelx    = cv2.Sobel(img_gray_crop, cv2.CV_64F, 1, 0, ksize=3)
sobely    = cv2.Sobel(img_gray_crop, cv2.CV_64F, 0, 1, ksize=3)
magnitude = np.sqrt(sobelx**2 + sobely**2)
direcao   = np.degrees(np.arctan2(sobely, sobelx))

# magnitude = mm.sobel(img_gray_crop) # ou, mais avec clip au lieu de norm255

mm.show(
```

```
    [img_gray_crop, norm255(np.abs(sobelx)), norm255(np.abs(sobely)),
     norm255(magnitude), norm255(direcao % 180)],
    titles=["Original", "Gx (bords vert.)", "Gy (bords horiz.)",
            "Magnitude | f|", "Direction  (mod 180°)"],
    cols=5
)
```

Figure 3.18: Opérateur de Sobel sur l'image du léopard : Gx détecte les bords verticaux, Gy détecte les bords horizontaux, et la magnitude | f| combine les deux, révélant tous les bords indépendamment de la direction.

3.6.3 Opérateur de Prewitt

L'opérateur de Prewitt est structurellement identique à Sobel, mais remplace la pondération gaussienne $\{1, 2, 1\}$ par des poids uniformes $\{1, 1, 1\}$:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} * f, \qquad G_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} * f \tag{3.22}$$

L'amplitude et la direction du gradient suivent les mêmes équations que Sobel (Équation 3.20 et Équation 3.21). La différence pratique est que Prewitt est légèrement plus sensible au bruit — le lissage perpendiculaire uniforme accorde moins de poids au pixel central de la ligne — mais il est plus simple sur le plan computationnel. Sur des images à faible bruit, les résultats sont équivalents.

```
Kx = np.array([[-1,0,1],[-1,0,1],[-1,0,1]], dtype=np.float32)
Ky = np.array([[-1,-1,-1],[0,0,0],[1,1,1]], dtype=np.float32)
f  = img_gray_crop.astype(np.float32)

mm.show(
    [img_gray_crop,
     norm255(np.abs(cv2.filter2D(f, -1, Kx))),
     norm255(np.abs(cv2.filter2D(f, -1, Ky))),
     mm.prewitt(img_gray_crop)],
    titles=["Original", "Gx", "Gy", "Magnitude | f|"],
    cols=4
)
```

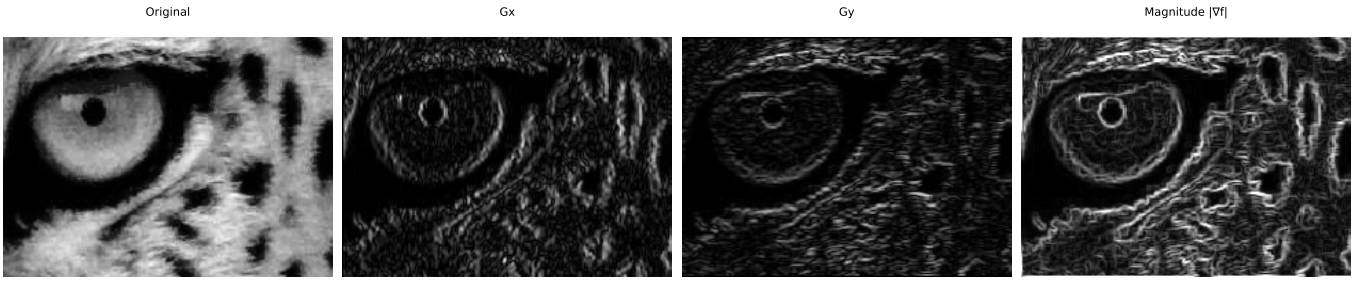


Figure 3.19: Opérateur de Prewitt : Gx, Gy et magnitude — comparable à Sobel, mais sans pondération gaussienne perpendiculaire.

3.6.4 *Unsharp Masking* (USM)

L'*Unsharp Masking* est une technique classique de renforcement de netteté issue de la photographie analogique, aujourd'hui largement utilisée dans les logiciels de retouche d'images. L'idée centrale est d'extraire les **composantes haute fréquence** de l'image (contours et détails) et de les additionner à l'originale avec un poids k :

Tableau 3.3: Étapes de l'*Unsharp Masking*.

Étape	Opération	Description
1	$\bar{f} = f * G_\sigma$	Lissage par gaussienne — conserve les basses fréquences
2	$m = f - \bar{f}$	Masque : différence = hautes fréquences (contours)
3	$g = f + k \cdot m$	Somme pondérée du masque à l'originale

En remplaçant l'étape 2 dans l'étape 3, on obtient l'expression compacte :

$$g = f + k\,(f - f * G_\sigma) = (1 + k)\,f - k\,(f * G_\sigma) \tag{3.23}$$

Le paramètre k contrôle l'intensité du renforcement :

- $k = 0$: aucun renforcement ($g = f$) ;
- $k = 1$: USM classique — double la contribution des hautes fréquences ;
- $k > 1$: *High Boost Filtering* — amplification au-delà du double, utile pour les images très floues.

> ⚠ Amplification du bruit
>
> L'USM ne distingue pas les contours du bruit — les deux sont des composantes haute fréquence. Pour un k élevé, le bruit présent dans l'image est amplifié en même temps que les contours. Il est donc recommandé d'appliquer un léger lissage avant l'USM sur les images bruitées, ou d'utiliser un σ petit dans la gaussienne.

Pour illustrer les étapes de l'USM, la Figure 3.20 applique la méthode à un *patch* 30×30 de l'image du léopard, en utilisant $\sigma = 1$ et $k = 1$. Initialement, l'image est lissée par un filtre gaussien. Ensuite, le masque haute fréquence est obtenu par la différence entre l'image originale et l'image lissée. Enfin, ce masque est ajouté à l'image originale, renforçant les contours et les détails. La figure présente les trois étapes du processus et le résultat final du rehaussement.

```
patch  = img_gray_crop[35:65, 45:75]
k = 1.0
sigma = 1.0

p = patch.astype(np.float32)

# Étape 1 : lissage gaussien
ksize   = int(6 * sigma + 1) | 1  # l'opérateur binaire garantit une taille impaire
p_suave = mm.gaussian(patch, ksize, sigma).astype(np.float32)

# Étape 2 : masque haute fréquence
mascara = p - p_suave

# Étape 3 : rehaussement
p_usm = np.clip(p + k * mascara, 0, 255).astype(np.uint8)

# p_usm = mm.usm(patch, k)  # ou, direct
```

```
print(f"Pixel central [1,1] : original={int(p[1,1])} lissé={p_suave[1,1]:.1f}"
      f" masque={mascara[1,1]:.1f} rehaussé={p_usm[1,1]}")

mm.show(
    [patch,
     p_suave.astype(np.uint8),
     np.clip(mascara + 128, 0, 255).astype(np.uint8),  # négatifs visibles
     p_usm],
    titles=["Patch original",
            f"Lissé ( ={sigma})",
            "Masque (haute fréquence, +128)",
            f"Rehaussé USM (k={k})"],
    cols=4, figsize=(12, 4)
)
```

```
Pixel central [1,1] : original=16 lissé=20.0 masque=-4.0 rehaussé=12
```

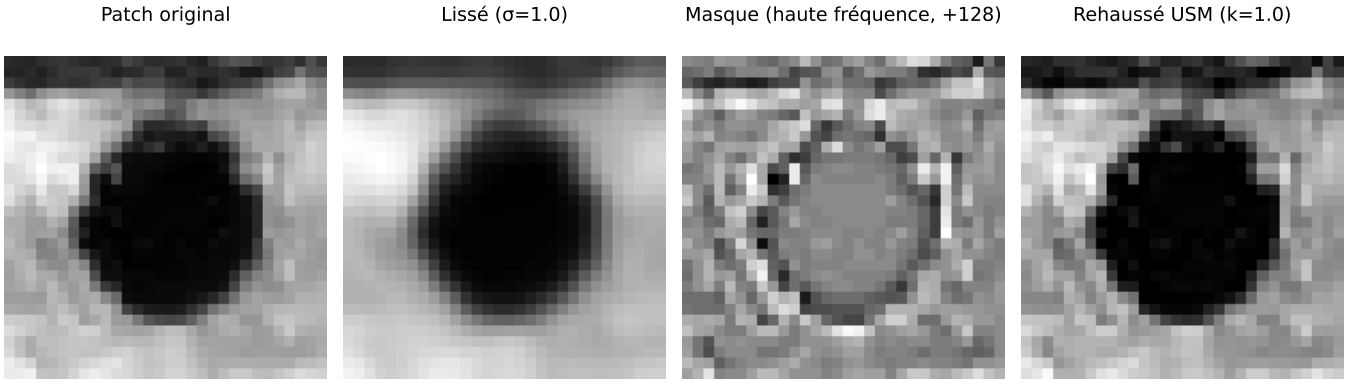


Figure 3.20: Rehaussement de netteté par *Unsharp Masking* sur l'image du léopard : l'image lissée est soustraite de l'originale pour générer le masque haute fréquence, qui est ensuite réintroduit pour accentuer les contours et les détails.

La Figure 3.21 applique la méthode USM à un recadrage plus large de l'image du léopard en utilisant $\sigma = 1$ et différentes valeurs du facteur de gain k. Dans tous les cas, le masque haute fréquence est obtenu par la différence entre l'image originale et sa version lissée par un filtre gaussien. Le paramètre k contrôle l'intensité du rehaussement : des valeurs plus faibles produisent une augmentation subtile de la netteté, tandis que des valeurs plus élevées renforcent progressivement les contours et les détails. On observe que, pour des valeurs élevées de k, des halos apparaissent autour des contours et le bruit présent dans l'image se trouve amplifié.

```
mm.show(
    [img_gray_crop,
     mm.usm(img_gray_crop, 0.5), mm.usm(img_gray_crop, 1.0),
     mm.usm(img_gray_crop, 3.0), mm.usm(img_gray_crop, 5.0),
     mm.usm(img_gray_crop, 8.0)],
    titles=["Original", "USM k=0,5", "USM k=1,0", "USM k=3,0", "USM k=5,0", "USM k=8,0"],
    cols=3
)
```

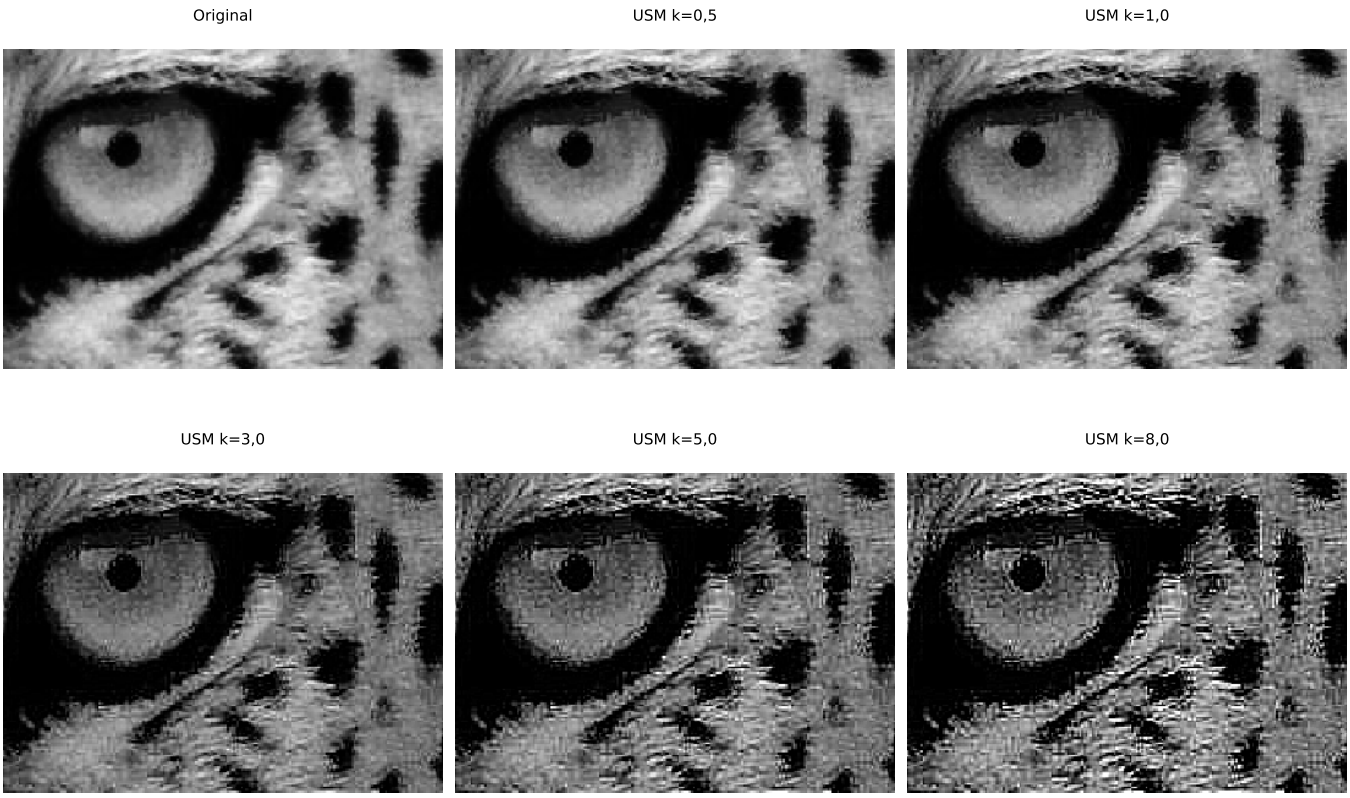


Figure 3.21: *Démasquage non net* sur l'image du léopard avec =1 et k de 0,5 à 8,0. Pour k>2 des halos apparaissent sur les bords et le bruit de fond ressort.

3.6.5 Détecteur de Canny

Canny combine quatre étapes en séquence — lissage gaussien, gradient de Sobel, suppression des non-maxima et hystérésis par double seuil — pour produire des contours **fins, binaires et connectés**. Contrairement à Sobel et Prewitt, le résultat n'est pas une carte de gradient continue, mais un masque où chaque pixel est un contour ou ne l'est pas.

Le paramètre central est la paire de seuils (T_{low}, T_{high}). Les pixels dont le gradient est supérieur à T_{high} sont des contours certains ; ceux en dessous de T_{low} sont écartés. Les pixels ambigus — entre les deux seuils — sont décidés par **hystérésis** : ils deviennent des contours s'ils sont connectés à un contour certain, et sont écartés dans le cas contraire. Cela évite à la fois la perte de segments faibles de contours réels et l'inclusion de bruit isolé. Une heuristique courante est $T_{high} = 3 \times T_{low}$.

```
mm.show(
    [img_gray_crop,
     mm.canny(img_gray_crop, 30,  90),
     mm.canny(img_gray_crop, 60,  180),
     mm.canny(img_gray_crop, 120, 240)],
    titles=["Original", "Canny (30/90)", "Canny (60/180)", "Canny (120/240)"],
    cols=4
)
```

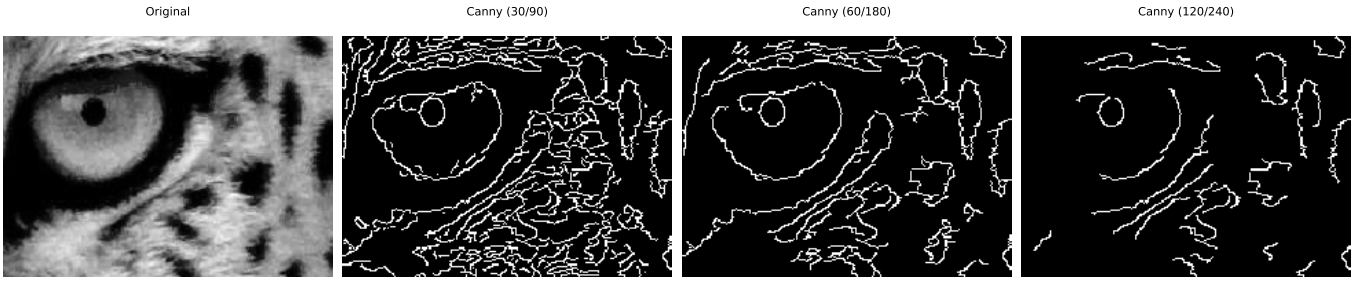


Figure 3.22: Détecteur de Canny avec différentes paires de seuils : des seuils bas capturent plus de bords (y compris le bruit) ; des seuils élevés ne conservent que les bords les plus forts.

Choix des seuils

Une heuristique courante est $T_{high} = 3 \times T_{low}$. Les valeurs typiques dépendent de la plage de gradient de l'image — `cv2.Canny` accepte des valeurs absolues dans $[0, 255]$. Pour des images à contraste variable, calculer les seuils à partir des percentiles de la magnitude de Sobel est plus robuste que des valeurs fixes.

3.7 Filtres d'Ordre : Filtre Médian

Les filtres d'ordre (*order-statistic filters*) remplacent le pixel central par la valeur d'un **centile** de la distribution des intensités du voisinage — contrairement aux filtres linéaires, qui calculent des combinaisons pondérées. Le plus important est le **filtre médian**.

3.7.1 Bruit Impulsif : Sel et Poivre

Le bruit **sel et poivre** (*salt-and-pepper noise*) remplace des pixels aléatoires par des valeurs extrêmes : 0 (poivre, noir) ou 255 (sel, blanc). Il est courant dans la transmission d'images avec des erreurs de bits et dans les caméras équipées de capteurs défectueux.

Pour comprendre pourquoi les filtres linéaires échouent, considérons un voisinage 3×3 où un seul pixel a été corrompu à 255 :

$$\text{voisinage} = \begin{bmatrix} 102 & 98 & 105 \\ 100 & \mathbf{255} & 97 \\ 103 & 99 & 101 \end{bmatrix}$$

Tableau 3.4: Moyenne vs. médiane avec un pixel corrompu. La médiane ignore la valeur aberrante ; la moyenne est décalée d'environ 40 niveaux.

Méthode	Calcul	Résultat
Moyenne	(102+98+…+255+…+101)/9	140
Médiane	{97,98,99,100,**101**,102,103,105,255}	101

Pourquoi les filtres de moyenne échouent-ils avec le bruit impulsif ?

La moyenne est sensible aux ***valeurs aberrantes*** — un seul pixel avec une valeur de 255 dans un voisinage de valeur 100 élève la sortie à 140, propageant le bruit dans l'image. La médiane, étant un **estimateur robuste**, sélectionne la valeur centrale de la distribution ordonnée, écartant naturellement les extrêmes sans aucun ajustement spécial.

L'exemple suivant illustre le comportement de la moyenne et de la médiane en présence d'un pixel corrompu par un bruit impulsif. On observe que la moyenne est fortement influencée par la valeur extrême (255), produisant une estimation éloignée des valeurs prédominantes du voisinage. Quant à la médiane, elle reste proche de la valeur originale de la région, mettant en évidence sa plus grande robustesse aux *valeurs aberrantes* et justifiant son utilisation dans la suppression du bruit sel et poivre.

```
# Exemple numérique : moyenne vs. médiane avec pixel corrompu
vizinhanca = np.array([102, 98, 105, 100, 255, 97, 103, 99, 101])
print(f"Voisinage : {sorted([int(x) for x in vizinhanca])}")
print(f"Moyenne :      {vizinhanca.mean():.1f}  (déplacée par la valeur aberrante 255)")
print(f"Médiane :    {int(np.median(vizinhanca))}    (ignore la valeur aberrante)")
print(f"Valeur originale :  ~100")
```

```
Voisinage : [97, 98, 99, 100, 101, 102, 103, 105, 255]
Moyenne :      117.8  (déplacée par la valeur aberrante 255)
Médiane :    101    (ignore la valeur aberrante)
Valeur originale :  ~100
```

La Figure 3.23 présente l'effet du bruit sel et poivre à différentes densités. Le bruit a été généré en remplaçant aléatoirement une fraction des pixels par des valeurs minimales (0, poivre) et maximales (255, sel). À mesure que la densité augmente de 2 % à 10 %, la quantité de pixels corrompus croît, rendant la dégradation visuelle plus évidente et rendant plus difficile la perception des détails de l'image.

```
def add_salt_pepper(img, prob=0.05, seed=42):
    """Ajoute du bruit sel et poivre avec une probabilité totale prob."""
    noisy = img.copy()
    rnd   = np.random.default_rng(seed).random(img.shape)
    noisy[rnd < prob / 2]      = 0    # poivre
    noisy[rnd > 1 - prob / 2] = 255  # sel
    n_corrompidos = int((rnd < prob/2).sum() + (rnd > 1-prob/2).sum())
    return noisy, n_corrompidos

probs = [0.02, 0.05, 0.10]
imgs_noise, titles_noise = [img_gray_crop], ["Original"]

for p in probs:
    noisy, n = add_salt_pepper(img_gray_crop, p)
    imgs_noise.append(noisy)
    titles_noise.append(f"Ruído {int(p*100)}%\n({n:,} pixels)")

mm.show(imgs_noise, titles=titles_noise, cols=4)
```

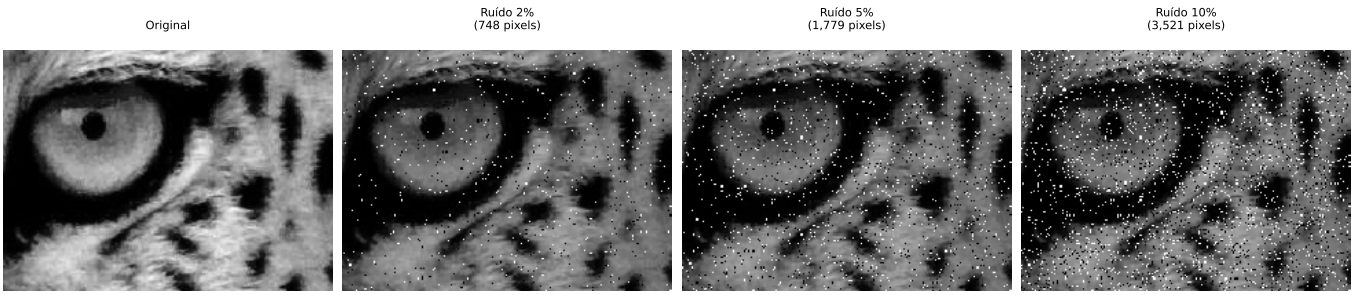


Figure 3.23: Bruit sel et poivre avec des densités croissantes (2 %, 5 %, 10 %). Le paramètre prob indique la fraction totale de pixels corrompus, moitié sel (255) et moitié poivre (0).

3.7.2 Filtre Médian

Le filtre médian remplace chaque pixel par la **valeur médiane** des pixels de son voisinage $n \times n$:

$$g(x, y) = \text{med}_{(s,t)\in\mathcal{V}_n}\{f(x + s, y + t)\} \tag{3.24}$$

La valeur médiane est celle qui occupe la position centrale lorsque les n^2 valeurs du voisinage sont triées. Pour une fenêtre 3×3 ($n^2 = 9$ pixels), la médiane est la 5 valeur de la séquence ordonnée.

Pour illustrer, considérons le même *patch* 5×5 avec un pixel corrompu artificiellement en $[1, 1]$:

```
patch    = img_gray[250:255, 250:255].copy()
corrompido = patch.copy()
corrompido[1, 1] = 255   # injecte un pixel de sel au centre

# Voisinage 3×3 centré sur [1,1]
viz_orig = sorted(patch[0:3, 0:3].ravel().tolist())
viz_corr = sorted(corrompido[0:3, 0:3].ravel().tolist())

print("Patch original :")
print(mm.drawImg(patch))
print("\nPatch avec pixel corrompu [1,1]=255 :")
print(mm.drawImg(corrompido))
print(f"\nVoisinage 3×3 original (trié) :   {viz_orig}")
print(f"Médiane originale :                       {viz_orig[4]}")
print(f"\nVoisinage 3×3 corrompu (trié) : {viz_corr}")
print(f"Médiane corrompue :                     {viz_corr[4]}  ← ignore le 255")
print(f"Moyenne corrompue :                  {np.mean(viz_corr):.1f}  ← faussée par le 255")
```

```
Patch 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


Patch avec pixel corrompu [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


Voisinage 3×3 original (trié) :   [91, 95, 102, 103, 106, 107, 107, 108, 108]
Médiane originale :                       106

Voisinage 3×3 corrompu (trié) : [91, 95, 102, 103, 106, 107, 108, 108, 255]
Médiane corrompue :                     106  ← ignore le 255
Moyenne corrompue :                  119.4  ← faussée par le 255
```

L'exemple confirme : même avec le pixel corrompu à 255, la médiane retourne la valeur centrale correcte — le *outlier* occupe la dernière position dans le tri et est écarté naturellement.

Parce qu'elle est basée sur le tri et non sur la somme, la médiane possède trois propriétés fondamentales qui la distinguent des filtres linéaires :

- **Robuste** au bruit impulsionnel — les outliers vont aux extrémités de la séquence triée et n'affectent pas la valeur centrale ;
- **Préservatrice des contours** — les transitions abruptes d'intensité sont conservées, car la médiane sélectionne une valeur qui existe déjà dans le voisinage, sans créer de nouveaux niveaux intermédiaires ;
- **Non linéaire** — elle ne peut pas être exprimée comme une convolution, donc `mm.conv` ne s'applique pas ; on utilise `cv2.medianBlur`.

La Figure 3.24 compare différentes techniques de suppression du bruit sel et poivre appliquées à une image contenant 10 % de pixels corrompus. Les filtres gaussien, moyen, médian, bilatéral et morphologique (chapitre suivant) ont été évalués, permettant ainsi d'observer le compromis entre la suppression du bruit et la préservation des détails. En général, les filtres de moyenne et gaussien réduisent le bruit, mais tendent à estomper les contours, tandis que le filtre médian offre de meilleures performances pour le bruit impulsionnel. Le filtre bilatéral préserve mieux les contours, et le filtre morphologique élimine une grande partie des pixels corrompus sans dégrader excessivement la structure de l'image.

```
# 10 % de bruit pour mettre en évidence les différences entre les filtres
noisy_5, _ = add_salt_pepper(img_gray_crop, prob=0.1)

#   Filtres
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)

# filtres morphologiques dans le chapitre suivant, mais déjà anticipés ici pour la comparaison
# B         = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
# f_morf    = cv2.morphologyEx(           # présenté dans le chapitre suivant
#                 cv2.morphologyEx(noisy_5, cv2.MORPH_OPEN,  B),
#                 cv2.MORPH_CLOSE, B)
B = mm.secross()  # élément structurant de boîte 3×3
f_morf     = mm.asf(noisy_5, 'OC', B)  # équivalent via morph

#   MSE
def mse(a, b):
    return float(np.mean((a.astype(np.float32) - b.astype(np.float32))**2))

print(f"{'Filtro':<22} {'MSE':>8}")
print("-" * 32)
pares = [("Gaussiano 5×5",    f_gauss),
         ("Média 5×5",        f_media),
         ("Mediana 3×3",      f_median3),
         ("Mediana 5×5",      f_median5),
         ("Bilateral",        f_bilat),
         ("Morf. open+close", f_morf)]
for nome, img in pares:
    print(f"{nome:<22} {mse(img_gray_crop, img):>8.2f}")

imgs   = [img_gray_crop, noisy_5,    f_gauss,        f_media,
          f_median3,     f_median5,  f_bilat,        f_morf]
titles = ["Original",    "Ruído 10%", "Gaussiano 5×5","Média 5×5",
          "Mediana 3×3", "Mediana 5×5", "Bilateral", "Morf. open+close"]

mm.show(
    imgs,
    titles=[f"Crop: {t}" for t in titles],
    cols=4, rows=2, figsize=(14, 8)
)
```

```
Filtro                     MSE
--------------------------------
Gaussiano 5×5            260.84
Média 5×5                872.57
Mediana 3×3               31.26
Mediana 5×5               65.86
Bilateral               1001.22
Morf. open+close          65.16
```

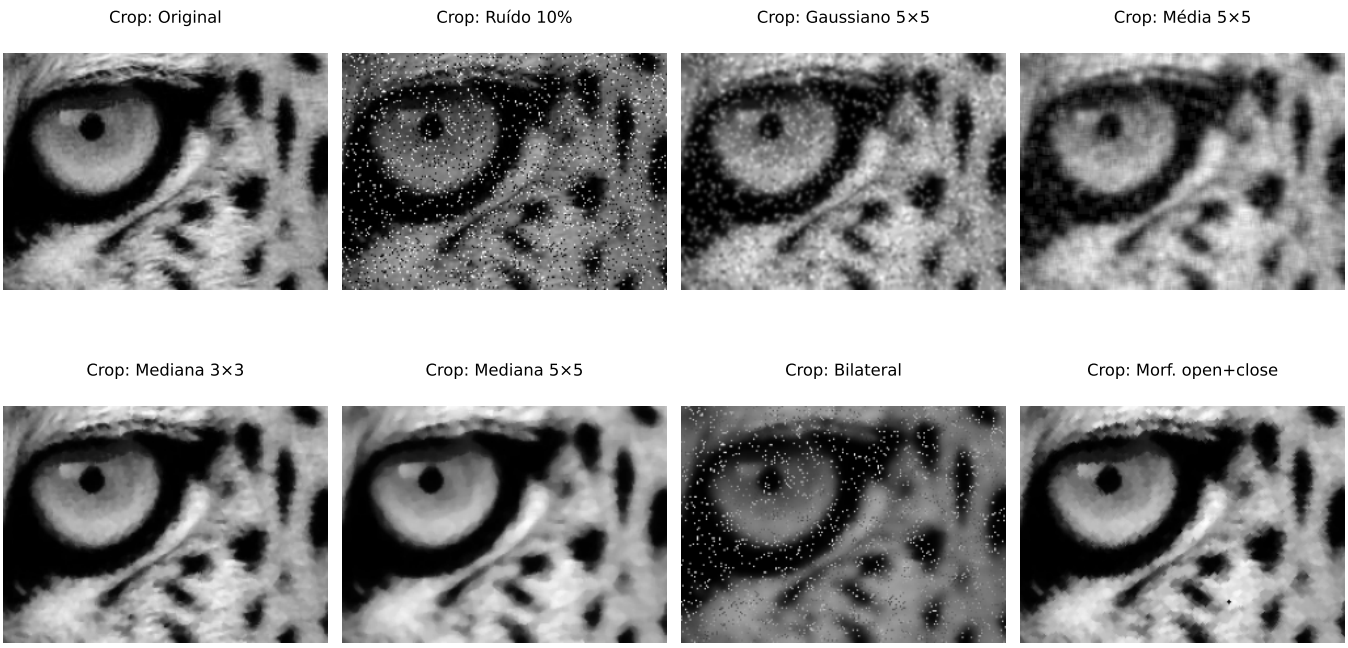


Figure 3.24: Comparaison de filtres pour la suppression du bruit sel et poivre (10 %) : Gaussien, Moyenne, Médiane, Bilatéral et Morphologique. Ligne supérieure : images complètes ; ligne inférieure : recadrage de la région d'intérêt.

3.8 Application pratique : Prétraitement pour la segmentation

En pratique, les techniques de ce chapitre sont rarement utilisées de manière isolée. Un ***pipeline* de prétraitement** typique combine plusieurs étapes en séquence, en s'adaptant au type d'image et à l'application. La Figure 3.25 illustre un *pipeline* complet :

1. **Égalisation d'histogramme (CLAHE) :** normalise le contraste indépendamment des conditions d'éclairage ;
2. **Filtre gaussien :** lisse le bruit d'acquisition sans détruire les contours ;
3. **Détection de contours (Sobel/Canny) :** extrait les structures pertinentes pour la segmentation.

> **ℹ L'ordre compte**
>
> L'ordre des opérations affecte le résultat final. En général : **(1) normalisation de l'intensité → (2) réduction du bruit → (3) rehaussement/segmentation**. Inverser l'ordre peut amplifier le bruit ou perdre les contours avant de les détecter.

```
# Étape 1 : CLAHE
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
img_clahe = clahe.apply(img_gray_crop)

# Étape 2 : Gaussiano
img_gauss = cv2.GaussianBlur(img_clahe, (5, 5), 0)

# Étape 3 : Canny
edges = cv2.Canny(img_gauss, 50, 150)

# Comparação : pipeline vs. Canny direto
edges_direct = cv2.Canny(img_gray_crop, 50, 150)

mm.show(
    [img_gray_crop, img_clahe, img_gauss, edges, edges_direct],
```

```
    titles=["Original", "1. CLAHE", "2. Gaussiano", "3. Canny (pipeline)", "Canny (direto)"],
    cols=5
)
```

Figure 3.25: *Pipeline* de pré-processamento : CLAHE → Gaussiano → Canny. Cada etapa prepara a imagem para a seguinte, resultando em bordas limpas e bem definidas.

3.9 Résumé

Dans ce chapitre, les principales techniques de traitement dans le domaine spatial ont été présentées, de la manipulation directe des pixels au filtrage par voisinage :

- **Opérations ponctuelles :** arithmétiques saturées (`mm.addm`, `mm.subm`) et logiques bit à bit (`mm.band`, `mm.bor`, `mm.bnot`) pour le recadrage de ROI et la combinaison d'images ; *alpha blending* (`mm.blend`) pour la fusion pondérée avec poids $\alpha \in [0, 1]$.
- **Histogramme :** fonction discrète de distribution des intensités ; visualisé avec `mm.histImg` et calculé avec `mm.hist` ; base pour le diagnostic tonal et pour les techniques d'égalisation et de spécification.
- **Égalisation :** redistribution automatique des intensités par la CDF (`mm.equalize`), avec variante adaptative CLAHE pour le contrôle local du contraste.
- **Spécification d'histogramme :** transfert du profil tonal d'une image de référence via le mappage inverse de la CDF — généralisation de l'égalisation pour des distributions arbitraires.
- **Corrélation et convolution :** mécanisme de fenêtre glissante implémenté dans `mm.conv` (`cv2.filter2D`) ; différenciés par la rotation de 180° du *kernel* — pertinent uniquement pour les kernels asymétriques.
- **Filtres de lissage :** moyenne (*kernel* uniforme, estompe les bords proportionnellement à la taille) et gaussien (pondération radiale, séparable, sans *ringing*, préserve mieux les bords).
- **Filtres de rehaussement :** laplacien (w_4/w_8, seconde dérivée isotrope), Sobel (gradient directionnel du premier ordre, avec magnitude $|\nabla f|$ et direction θ) et *Unsharp Masking* (amplification des hautes fréquences avec paramètre k).
- **Filtre médian :** non linéaire, robuste aux outliers, préserve les bords — supérieur aux filtres linéaires pour le bruit sel et poivre.
- **Pipeline pratique :** enchaînement CLAHE → Gaussien → Canny comme stratégie de prétraitement ; `mm.drawImgKernel` pour la visualisation didactique de la fenêtre glissante.

Le Chapitre 4 abordera la **morphologie mathématique** (érosion, dilatation, ouverture et fermeture), en explorant en profondeur les fonctions `mm.ero` et `mm.dil` de la bibliothèque `morph.py`. Ensuite, le Chapitre 5 présentera le **traitement dans le domaine fréquentiel**, avec un accent sur la Transformée de Fourier et les techniques de filtrage spectral.

3.10 🤖 Utilisation de Gemini Notebook comme tuteur complémentaire

Dans cette édition, nous encourageons l'utilisation de **Gemini Notebook** comme outil d'apprentissage complémentaire. Cet outil d'IA utilise exclusivement les documents fournis par l'auteur comme base de connaissances, garantissant des réponses cohérentes avec le contenu du livre — y compris les fonctions de la bibliothèque `morph.py` et les expériences réalisées dans ce chapitre.

Pour chaque chapitre, nous avons préparé un projet spécifique sur la plateforme avec le PDF du chapitre, les *notebooks* et les supports auxiliaires. Nous suggérons d'explorer en particulier :

- **Guide d'étude :** résumé structuré des concepts, idéal pour la révision avant les examens ;
- **Conversation :** posez vos questions sur l'égalisation, la convolution, les filtres et les pipelines directement au tuteur ;
- **Questions fréquentes :** questions typiques sur la différence entre moyenne et médiane, USM, Laplacien vs. Sobel.

🎓 Étudiez avec le tuteur intelligent

Pour interagir avec le contenu de ce chapitre, accédez au *lien* suivant. L'environnement contient des supports pédagogiques sous différents formats, générés à partir du **PDF** du chapitre. Sur la plateforme, explorez particulièrement les options **Guide d'étude** et **Conversation** pour approfondir votre compréhension.
🚀 ACCÉDER À GEMINI NOTEBOOK : CHAPITRE 03

🌐 **Langue et langage de programmation**

Le projet de ce chapitre dans Gemini Notebook a été construit uniquement avec le texte en **portugais** et les exemples de code en **Python**. Si vous étudiez à partir de l'édition en anglais ou en français, ou que vous suivez le parcours en C++, les réponses du tuteur peuvent ne pas correspondre exactement à la version que vous lisez.

⚠️ **Avertissement concernant le contenu généré par l'IA**

L'IA est une alliée puissante dans les études, mais le contenu généré peut contenir des **erreurs ou des imprécisions**. Consultez toujours des **livres, articles scientifiques et autres sources académiques fiables** pour valider les informations. Dans la mesure du possible, exécutez les exemples pratiques fournis dans ce chapitre pour vérifier les résultats.

3.11 Liste d'exercices

1. **(10%)** Expliquez la différence entre **convolution** et **corrélation croisée**. Pour quels types de *noyaux* les résultats sont-ils identiques ? Donnez un exemple de *noyau* asymétrique (comme Sobel G_x) et montrez numériquement que les résultats diffèrent en l'appliquant au patch 5×5 du chapitre de deux manières.

2. **(15%)** Considérez une image 5×5 avec des intensités concentrées entre les niveaux 3 et 5 (faible contraste, 3 bits). Appliquez manuellement l'algorithme d'égalisation de la Table 3.1, en remplissant toutes les colonnes du tableau (k, $h[k]$, $p[k]$, cdf$[k]$, lut$[k]$). Vérifiez le résultat avec `mm.equalize`.

3. **(15%)** En utilisant `mm.conv`, appliquez le filtre moyenneur avec des noyaux de tailles 3×3, 9×9 et 21×21 à l'image du mandrill. Pour chaque version, calculez le **PSNR** (*Peak Signal-to-Noise Ratio*) par rapport à l'originale :

$$\mathrm{PSNR} = 10\log_{10}\left(\frac{255^2}{\mathrm{MSE}}\right), \quad \mathrm{MSE} = \frac{1}{MN}\sum_{i,j}(f-g)^2$$

   Tracez le PSNR en fonction de la taille du noyau et expliquez ce que la chute progressive indique sur la relation entre le lissage et la perte d'information.

4. **(15%)** En utilisant `add_salt_pepper` avec une densité de 5%, appliquez et comparez : (a) `mm.conv` avec moyenne 3×3, (b) `cv2.GaussianBlur` avec $\sigma = 1$, (c) `cv2.medianBlur` avec fenêtre 3×3 et (d) `cv2.medianBlur` avec fenêtre 5×5. Affichez les images avec `mm.show` dans une grille 2×4 (ligne 1 : images, ligne 2 : histogrammes via `mm.histImg`). Expliquez pourquoi la médiane surpasse les filtres linéaires en utilisant l'argument de la Table 3.4..

5. **(15%)** Implémentez `mm.conv0` en utilisant uniquement des opérations NumPy vectorisées — sans boucles Python et sans `cv2.filter2D` — avec l'opérateur de *stride tricks* (`np.lib.stride_tricks.sliding_win` Comparez le résultat et le temps d'exécution avec `mm.conv0` (boucles) et `mm.conv` (cv2) pour des noyaux 3×3 et 15×15 sur l'image du mandrill.
6. **(15%)** Appliquez le *Unsharp Masking* avec $\sigma = 1$ et $k \in \{0.5, 1.0, 2.0, 4.0\}$ en utilisant la fonction `usm` du chapitre. Pour chaque valeur de k : (a) calculez la différence absolue $|g - f|$, (b) affichez les images et les différences avec `mm.show`, et (c) tracez l'histogramme des différences avec `mm.histImg`. Identifiez à partir de quel k les artefacts (halos et amplification du bruit) deviennent visuellement inacceptables.
7. **(15%)** Choisissez une image de radiographie ou de tomographie disponible publiquement (ex. : via `mm.read` à partir d'une URL) et concevez un pipeline de prétraitement avec au moins 4 étapes séquentielles, en justifiant chaque choix à partir des concepts du chapitre. Affichez avec `mm.show` dans une grille : l'image originale, chaque étape intermédiaire et le résultat final avec leurs histogrammes (`mm.histImg`).

Références du chapitre

La base théorique de ce chapitre s'appuie sur les ouvrages suivants :

- Gonzalez (2018) pour les concepts d'opérations d'intensité, d'histogramme, de convolution et de filtrage spatial.
- Szeliski (2022) pour la vision par ordinateur et les applications pratiques du filtrage.
- Bradski (2008) pour l'implémentation pratique avec OpenCV et `morph.py`.

Exécuter Colab | Ouvrir GitHub

3.12 💻 Partie Pratique avec Exercices de Programmation

🎯 Objectif de ce cahier

Ce cahier permet de développer, valider, organiser et tester des solutions d'**Exercices de Programmation (EPs)** dans des environnements interactifs, tels que Colab, avec les mêmes cas de test que Moodle, en y copiant uniquement au moment d'enregistrer la note officielle.

Téléchargement

Téléchargez `morph.py` et `testsuite.py` en exécutant la cellule ci-dessous :

```
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
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0 | TestSuite : 1.1.2
```

Exécution des tests

Pour évaluer les tests, exécutez `TestSuite("EP03_01.extensão").run()` dans une nouvelle cellule, en remplaçant l'extension par celle du langage utilisé (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). Le système télécharge

les cas de test depuis GitHub, exécute le programme et calcule la note automatiquement.

Pour tester du code Python directement, sans enregistrer de fichier, utilisez `run_code(codigo)` en passant le code sous forme de *chaîne de caractères* dans une variable `codigo` :

```
codigo = """
from morph import mm
# ... votre code ici ...
"""
TestSuite("EP03_01").run_code(codigo)
```

3.12.1 EP03_01 ➕ Addition Saturée de Constante

Dans les systèmes de surveillance vidéo, les caméras situées dans des environnements à éclairage variable produisent des images sous-exposées. Le réglage de la luminosité par **addition saturée d'une constante** est l'opération la plus simple pour une correction immédiate, étant appliquée en temps réel sur les *puces* des caméras embarquées et dans les *pipelines* de pré-traitement des robots mobiles.

Voir dans Figure 3.26 une simulation de ce EP.

3.12.1.1 📋 Directives d'Implémentation

1. **Dimensions :** Lire les entiers L (lignes) et C (colonnes).
2. **Constante :** Lire l'entier k (valeur à ajouter).
3. **Données :** Lire les valeurs entières de la matrice originale ligne par ligne.
4. **Mappage :** Pour chaque pixel p, calculer la nouvelle valeur selon l'équation :

$$p' = \text{clip}(p + k)$$

5. **Sortie :** Afficher la matrice résultante avec les dimensions $L \times C$.

3.12.1.2 📌 Contraintes Computationnelles

- **Saturation (*Clipping*) :** Les valeurs doivent être confinées à l'intervalle $[0, 255]$:

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

- **Type :** Le résultat final doit être entier (sans décimales).
- **k peut être négatif :** les valeurs négatives assombrissent l'image ; les valeurs positives l'éclaircissent.

3.12.1.3 🧠 Fondement Théorique

Paramètre	Type	Impact Visuel
$k > 0$	Entier	Éclaircit l'image ; les pixels proches de 255 saturent en blanc
$k < 0$	Entier	Assombrit l'image ; les pixels proches de 0 saturent en noir
$k = 0$	Entier	Image inchangée

3.12.1.4 📦 Spécification d'Entrée et de Sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Entier k.

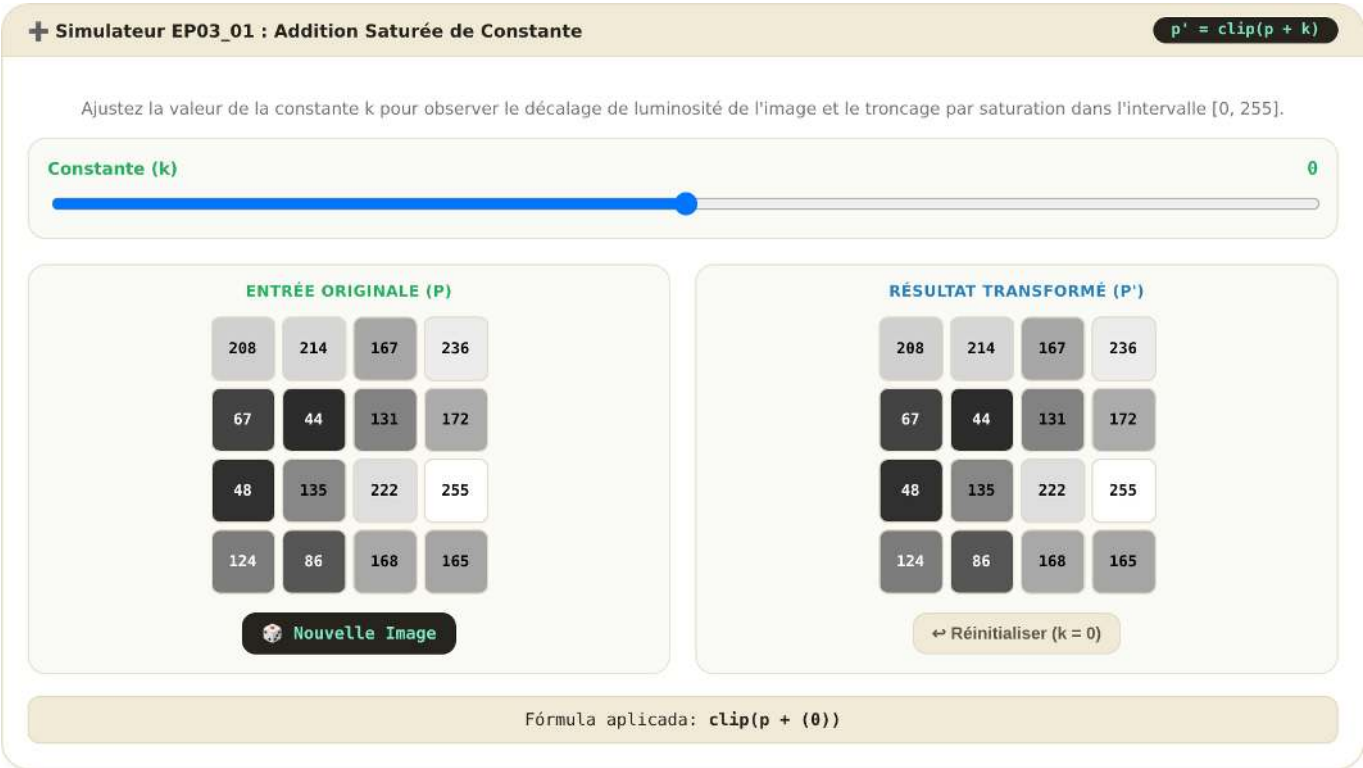
- Lignes suivantes : Éléments entiers de la matrice originale.

Sortie :

- Matrice transformée en L lignes et C colonnes, valeurs entières séparées par des espaces.

3.12.1.5 📌 Exemples

Entrée	Sortie	Observation
2 3 50 0 100 200 210 240 255	50 150 250 255 255 255	Saturation à 255 sur les pixels élevés
1 4 -30 0 20 200 255	0 0 170 225	Saturation à 0 sur les pixels bas



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap03/sim-ep0301-adicao.html

Figure 3.26: Simulateur EP03_01 : Addition saturée de constante (p' = clip(p + k))

```
%%writefile EP03_01.py
# Code Python
```

```
Overwriting EP03_01.py
```

```
TestSuite("EP03_01.py").run()
```

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

3.12.2 EP03_02 🔀 Fusion *Alpha* de Deux Images

En médecine nucléaire, des images de différentes modalités (tomodensitométrie et imagerie par résonance magnétique) sont fusionnées pour faciliter le diagnostic. Le **mélange pondéré** (*alpha blending*) est l'opération fondamentale de ce processus, permettant au radiologue de contrôler interactivement le poids de chaque modalité dans l'image affichée.

Voir dans Figure 3.27 une simulation de ce EP.

3.12.2.1 📋 Directives d'Implémentation

1. **Dimensions :** Lire les entiers L (lignes) et C (colonnes).
2. **Paramètre :** Lire la valeur réelle $\alpha \in [0, 1]$.
3. **Données :** Lire les valeurs entières de la matrice f_1 (image 1), puis celles de la matrice f_2 (image 2).
4. **Mappage :** Pour chaque position (i, j), calculer :

$$g(i, j) = \text{clip}\left(\text{round}\left(\alpha \cdot f_1(i, j) + (1 - \alpha) \cdot f_2(i, j)\right)\right)$$

5. **Sortie :** Afficher la matrice résultante $L \times C$.

3.12.2.2 📌 Contraintes de Calcul

- **Arrondi :** Appliquer `round` avant la conversion en entier.
- **Saturation :** Confiner à l'intervalle $[0, 255]$ avec $\text{clip}(x) = \max(0, \min(255, x))$.
- **Opération en virgule flottante :** Effectuer l'opération en virgule flottante avant d'arrondir.

3.12.2.3 🧠 Fondement Théorique

Valeur de α	Résultat
$\alpha = 1.0$	Uniquement f_1
$\alpha = 0.5$	Moyenne arithmétique de f_1 et f_2
$\alpha = 0.0$	Uniquement f_2

3.12.2.4 📦 Spécification des Entrées et Sorties (VPL)

Entrée :

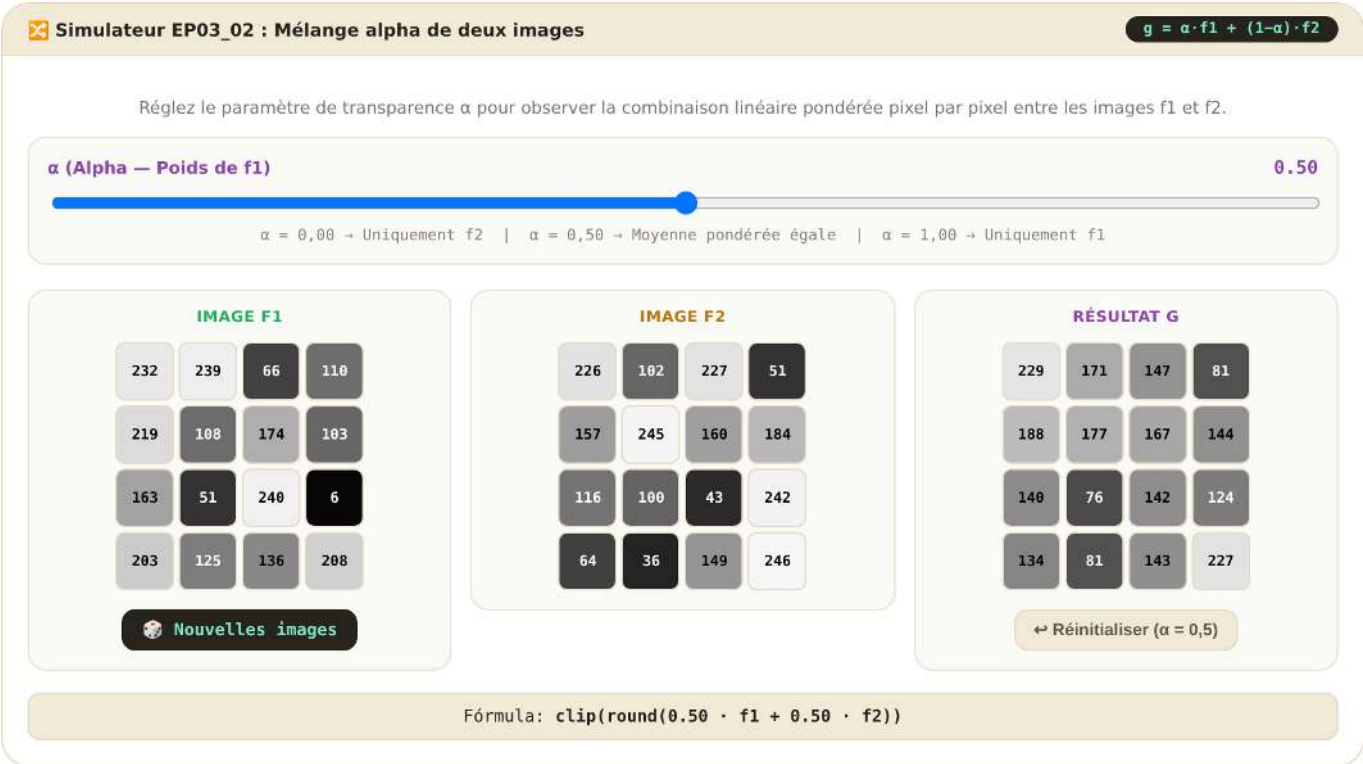
- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Réel α.
- Lignes suivantes : Éléments de f_1 (L lignes avec C valeurs chacune).
- Lignes suivantes : Éléments de f_2 (L lignes avec C valeurs chacune).

Sortie :

- Matrice résultante $L \times C$.

3.12.2.5 📌 Exemples

Entrée	Sortie	Observation
1 3 0.5 0 100 200 100 200 50	50 150 125	Moyenne entre les deux images
1 3 1.0 10 20 30 90 80 70	10 20 30	Uniquement f_1 (alpha=1)



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap03/sim-ep0302-blending.html

Figure 3.27: Simulateur EP03_02 : Mélange Alpha de Deux Images (g = · f1 + (1−) · f2)

```
%%writefile EP03_02.py
# Code Python
```

```
Overwriting EP03_02.py
```

```
TestSuite("EP03_02.py").run()
```

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

3.12.3 EP03_03 🎭 Inversion d'image (négatif photographique)

En radiologie, les images de rayons X sont traditionnellement visualisées en négatif : les os apparaissent en noir sur fond blanc. L'opération de **négatif photographique** est appliquée de manière courante dans les PACS (*Picture Archiving and Communication Systems*) pour faciliter la détection des fractures et des densités osseuses.

Voir dans Figure 3.28 une simulation de cet EP.

3.12.3.1 📋 Directives d'implémentation

1. **Dimensions :** Lire les entiers L (lignes) et C (colonnes).
2. **Données :** Lire les valeurs entières de la matrice originale.
3. **Mappage :** Pour chaque pixel p, calculer le négatif :

$$p' = 255 - p$$

4. **Sortie :** Afficher la matrice résultante $L \times C$.

3.12.3.2 📌 Contraintes informatiques

- **Aucun *clipping* nécessaire :** Le résultat de $255 - p$ avec $p \in [0, 255]$ est toujours $\in [0, 255]$.
- **Type entier :** La sortie doit être des valeurs entières.
- **Équivalence logique :** L'opération est identique au `NOT` bit à bit (`mm.bnot`) sur les images en 8 bits.

3.12.3.3 🧠 Fondements théoriques

Pixel original p	Pixel négatif p'	Observation
0 (noir)	255 (blanc)	Inversion totale
128 (gris moyen)	127 (gris moyen)	Valeur centrale
255 (blanc)	0 (noir)	Inversion totale

3.12.3.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Lignes suivantes : Éléments entiers de la matrice originale.

Sortie :

- Matrice négative en L lignes et C colonnes.

3.12.3.5 📌 Exemples

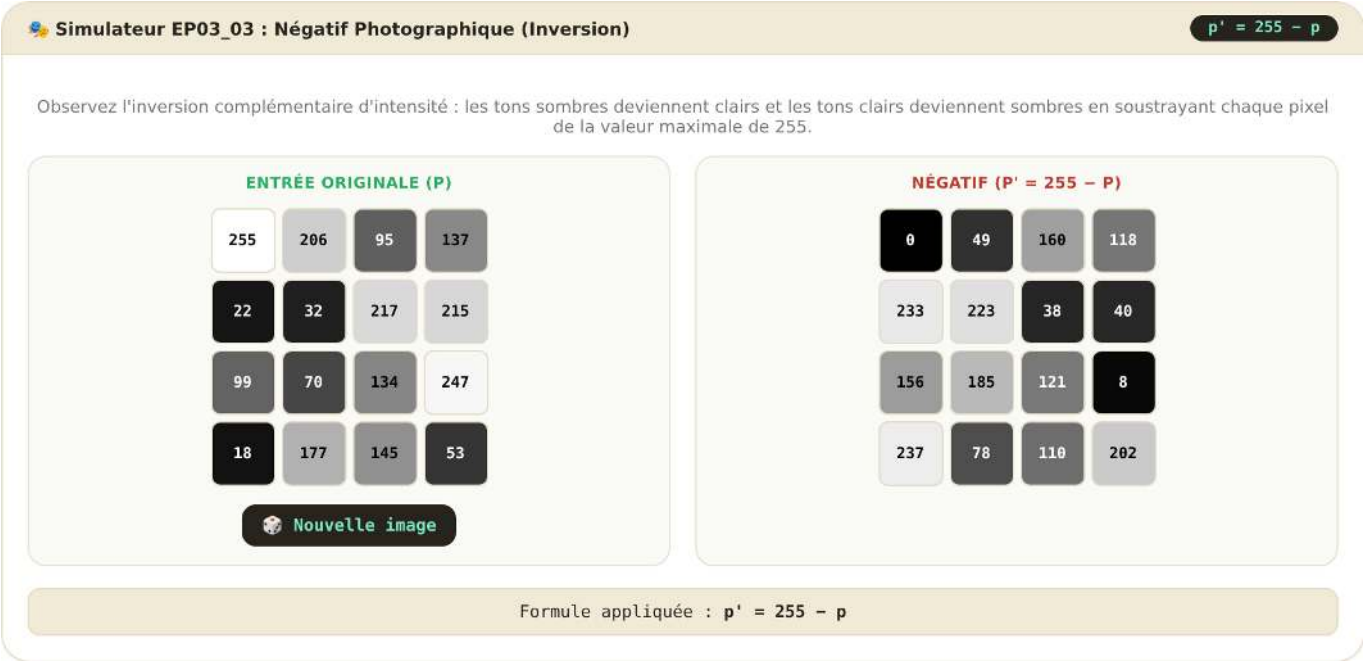
Entrée	Sortie	Observation
140 128 200 255	255 127 55 0	Inversion de chaque pixel
2 2 10 20 30 40	245 235225 215	Matrice 2x2 inversée

```
%%writefile EP03_03.py
# Code 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.fr/cap03/sim-ep0303-inversao.html

Figure 3.28: Simulador EP03_03 : Inversion d'image — Négatif photographique (p' = 255 − p)

3.12.4 EP03_04 📊 Égalisation d'histogramme (L bits)

Dans les images satellites de télédétection, la variation de l'éclairage au cours de la journée produit des images à faible contraste. L'**égalisation d'histogramme** est appliquée automatiquement sur des satellites comme Landsat pour redistribuer les tons, révélant des détails de végétation, de relief et de zones urbaines invisibles dans l'image originale.

Voir dans Figure 3.29 une simulation de cet EP.

3.12.4.1 📋 Directives d'implémentation

1. **Dimensions :** Lire les entiers L (lignes), C (colonnes) et B (nombre de bits, avec $L_{\max} = 2^B$).
2. **Données :** Lire la matrice de pixels f avec des valeurs dans $[0, 2^B - 1]$.
3. **Histogramme :** Calculer $h[k]$ = nombre de pixels avec l'intensité k, pour $k = 0 \dots 2^B - 1$.
4. **Probabilité :** $p[k] = h[k]/(L \cdot C)$.
5. **CDF :** $\mathrm{cdf}[k] = \sum_{j=0}^{k} p[j]$; fonction de distribution cumulée.
6. **LUT :** $\mathrm{lut}[k] = \mathrm{round}\,(\mathrm{cdf}[k] \cdot (2^B - 1))$; *Look-Up Table* (table de consultation).
7. **Application :** $g[i, j] = \mathrm{lut}[f[i, j]]$.
8. **Sortie :** Afficher la matrice égalisée $L \times C$.

3.12.4.2 📌 Contraintes computationnelles

- **Arrondi :** Utiliser l'arrondi mathématique (`round`) dans la LUT.
- **Bits :** Le nombre de niveaux est 2^B (ex. : $B = 3 \Rightarrow 8$ niveaux, $B = 8 \Rightarrow 256$ niveaux).
- **CDF cumulée :** $\mathrm{cdf}[k] = \sum_{j=0}^{k} p[j]$, avec $\mathrm{cdf}[2^B - 1] = 1.0$.

3.12.4.3 🧠 Fondement théorique

Étape	Opération	Formule
1	Histogramme	$h[k] \leftarrow$ nombre de pixels avec l'intensité k
2	Probabilité	$p[k] = h[k]/(L \cdot C)$

Étape	Opération	Formule
3	CDF	$\text{cdf}[k] = \sum_{j=0}^{k} p[j]$
4	LUT	$\text{lut}[k] = \text{round}(\text{cdf}[k] \cdot (2^B - 1))$
5	Application	$g[i,j] = \text{lut}[f[i,j]]$

3.12.4.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Entier B (nombre de bits).
- Lignes suivantes : Éléments entiers de la matrice.

Sortie :

- Matrice égalisée en L lignes et C colonnes.

3.12.4.5 📌 Exemples

Entrée	Sortie	Observation
5 5 3 3 4 2 3 4 4 3 3 4 3 2 3 4 3 2 3 4 3 2 3 4 3 2 3 4	5 7 1 5 7 7 5 5 7 5 1 5 7 5 1 5 7 5 1 5 7 5 1 5 7	Exemple 3 bits du chapitre
1 4 3 0 0 7 7	0 0 7 7	Histogramme bimodal extrême

```
%%writefile EP03_04.py
# Code Python
```

```
Overwriting EP03_04.py
```

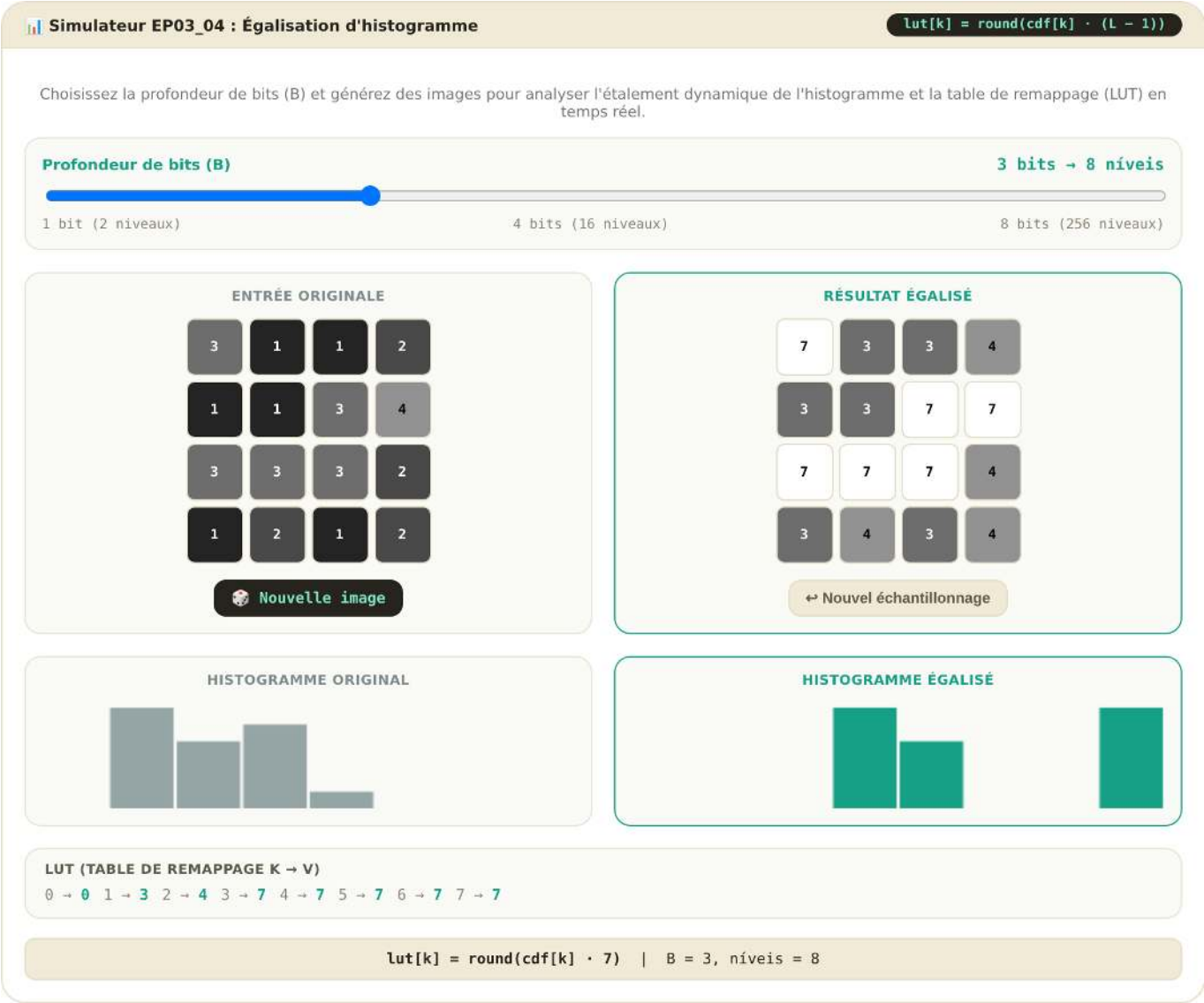
```
TestSuite("EP03_04.py").run()
```

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

3.12.5 EP03_05 🔲 Application du masque binaire ET

Dans les systèmes d'inspection industrielle par vision par ordinateur, il est nécessaire d'isoler les régions d'intérêt (ROI) dans les images de pièces afin de vérifier les défauts de fabrication. L'opération **ET bit à bit avec un masque binaire** est le mécanisme fondamental pour découper exactement la zone d'inspection, en mettant à zéro tous les pixels en dehors de celle-ci.

Voir dans Figure 3.30 une simulation de cet EP.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap03/sim-ep0304-equalizacao.html

Figure 3.29: Simulateur EP03_04 : Égalisation d'histogramme (Niveaux L = 2^B)

3.12.5.1 📋 Directives d'implémentation

1. **Dimensions :** Lire les entiers L (lignes) et C (colonnes).
2. **Données :** Lire la matrice de pixels f (valeurs $\in [0, 255]$).
3. **Masque :** Lire la matrice binaire m (valeurs : uniquement 0 ou 255).
4. **Mappage :** Pour chaque pixel (i, j), appliquer le ET bit à bit :

$$g(i, j) = f(i, j) \text{ ET } m(i, j)$$

où $255 =$ `11111111` et $0 =$ `00000000` en binaire.

5. **Sortie :** Afficher la matrice résultante $L \times C$.

3.12.5.2 📌 Contraintes de calcul

- **ET avec 255 :** p ET $255 = p$ (tous les bits sont préservés).
- **ET avec 0 :** p ET $0 = 0$ (tous les bits sont mis à zéro).
- **Masque :** Les seules valeurs possibles dans le masque sont 0 et 255.
- **Implémentation :** En Python, le ET bit à bit entre entiers utilise l'opérateur `&`.

3.12.5.3 🧠 Fondement théorique

Pixel f	Masque m	Résultat f ET m
n'importe quel v	255 (`11111111`)	v (préservé)
n'importe quel v	0 (`00000000`)	0 (mis à zéro)

3.12.5.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

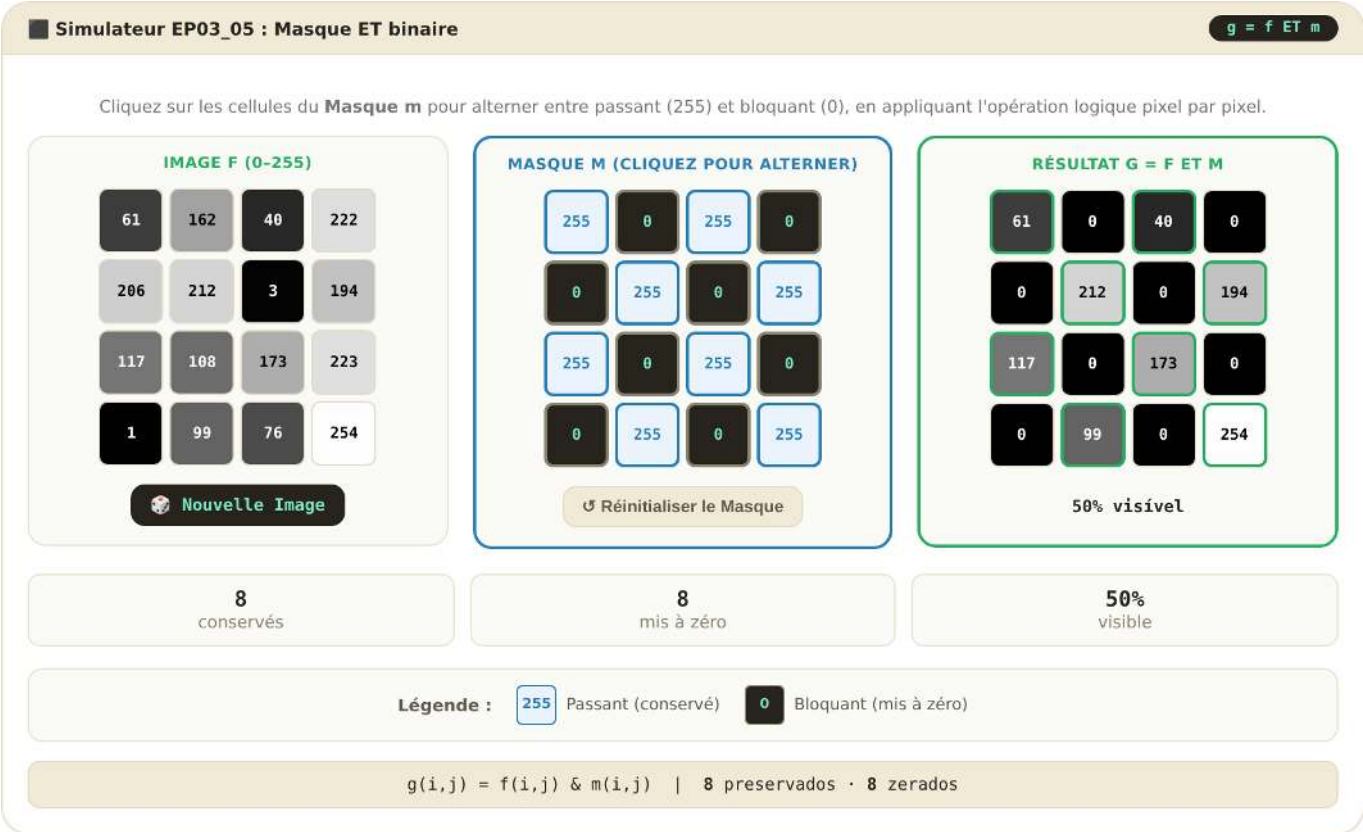
- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Lignes suivantes : Éléments de f (L lignes).
- Lignes suivantes : Éléments de m (L lignes avec valeurs 0 ou 255).

Sortie :

- Matrice résultante $L \times C$.

3.12.5.5 📌 Exemples

Entrée	Sortie	Observation
2 3 100 150 200 50 80 120 255 255 0 0 255 255	100 150 0 0 80 120	Le masque sélectionne la région
1 4 10 20 30 40 255 0 255 0	10 0 30 0	Alternance préservé/mis à zéro



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap03/sim-ep0305-mascara.html

Figure 3.30: Simulateur EP03_05: Application de masque AND binaire

```
%%writefile EP03_05.py
# Code Python
```

```
Overwriting EP03_05.py
```

```
TestSuite("EP03_05.py").run()
```

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

3.12.6 EP03_06 Filtre de moyenne avec *kernel* N×N

Dans les caméras de véhicules autonomes, les images capturées sous la pluie ou le brouillard présentent un bruit gaussien. Le **filtre de moyenne** est largement utilisé pour sa réduction en temps réel, étant implémenté directement dans le **ISP** (*Image Signal Processor*) des capteurs **CMOS** (*Complementary Metal-Oxide-Semiconductor*).

Les capteurs CMOS sont les capteurs d'image utilisés dans la plupart des caméras modernes (*smartphones*, *webcams*, caméras automobiles, etc.). Ils convertissent la lumière en signaux électriques, et l'ISP traite ces signaux en temps réel — en appliquant des opérations telles que la réduction du bruit, la balance des blancs et d'autres ajustements d'image.

Voir dans Figure 3.31 une simulation de cet EP.

3.12.6.1 📋 Directives d'implémentation

1. **Dimensions :** Lire les entiers L (lignes), C (colonnes) et N (taille du *kernel*, toujours impair).
2. **Données :** Lire la matrice de pixels f.
3. **Filtre de moyenne :** Pour chaque pixel (i, j) **interne** (sans bordures), calculer :

$$g(i,j) = \text{round}\left(\frac{1}{N^2}\sum_{s=-(r)}^{r}\sum_{t=-(r)}^{r} f(i+s,\, j+t)\right), \quad r = \lfloor N/2 \rfloor$$

4. **Traitement des bords :** Les pixels sur le bord (où la fenêtre $N \times N$ dépasse les limites) doivent être **copiés directement** depuis l'original sans modification.
5. **Sortie :** Afficher la matrice résultante $L \times C$.

3.12.6.2 📌 Contraintes de calcul

- **Rayon :** $r = \lfloor N/2 \rfloor$ (moitié du *kernel*, entier).
- **Pixels internes :** (i,j) avec $r \leq i < L - r$ et $r \leq j < C - r$.
- **Arrondi :** Utiliser l'arrondi mathématique avant de convertir en entier.
- **Sans *clipping* :** La moyenne des valeurs $\in [0, 255]$ reste dans $[0, 255]$.

3.12.6.3 🧠 Fondements théoriques

Taille N	Coefficient	Pixels dans la fenêtre	Effet
3	$1/9 \approx 0.111$	9	Lisse
5	$1/25 = 0.04$	25	Moyen
7	$1/49 \approx 0.020$	49	Fort

3.12.6.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

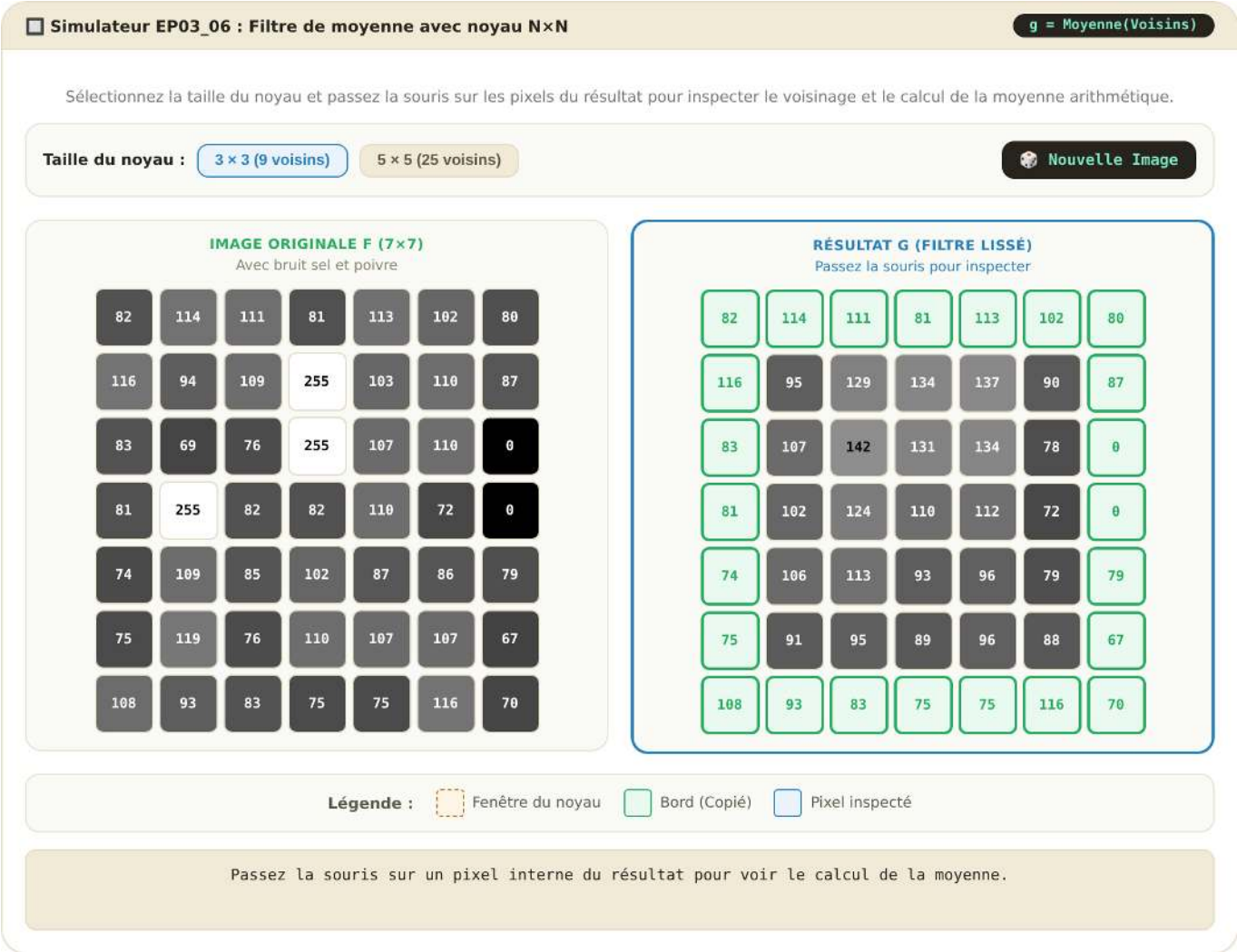
- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Entier N (impair, $N \geq 3$).
- Lignes suivantes : Éléments de la matrice originale.

Sortie :

- Matrice filtrée $L \times C$.

3.12.6.5 📌 Exemples

Entrée	Sortie	Observation
3 3 3 10 20 30 40 50 60 70 80 90	10 20 30 40 50 60 70 80 90	Bordure uniquement (3×3 = bordure totale)
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	Pixel isolé : tous les 9 pixels internes dont la fenêtre 3×3 inclut la valeur 100 reçoivent round(100/9)=11



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap03/sim-ep0306-media.html

Figure 3.31: Simulateur EP03_06: Filtre de Moyenne avec Noyau N×N

```
%%writefile EP03_06.py
# Code Python
```

```
Overwriting EP03_06.py
```

```
TestSuite("EP03_06.py").run()
```

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

3.12.7 EP03_07 🔍 Opérateur Laplacien (w4) pour le Rehaussement des Contours

Dans les tomographies à haute résolution, la netteté des contours entre les tissus est critique pour le diagnostic. L'**opérateur Laplacien** est largement utilisé dans les *pipelines* de prétraitement d'images médicales pour rehausser automatiquement les contours anatomiques avant la segmentation, évitant ainsi une intervention manuelle du radiologiste.

Voir dans Figure 3.32 une simulation de cet EP.

3.12.7.1 📋 Directives d'Implémentation

1. **Dimensions :** Lire les entiers L (lignes) et C (colonnes).
2. **Données :** Lire la matrice de pixels f.
3. **Laplacien (w4) :** Pour chaque pixel **interne** (i, j) avec $1 \leq i < L - 1$, $1 \leq j < C - 1$, calculer :

$$\nabla^2 f(i, j) = f(i-1, j) + f(i+1, j) + f(i, j-1) + f(i, j+1) - 4 \cdot f(i, j)$$

4. **Rehaussement :** Calculer l'image rehaussée :

$$g(i, j) = \text{clip}(f(i, j) - \nabla^2 f(i, j))$$

5. **Bordure :** Les pixels en bordure sont copiés directement : $g(i, j) = f(i, j)$.
6. **Sortie :** Afficher la matrice rehaussée $L \times C$.

3.12.7.2 📌 Contraintes Computationnelles

- ***Noyau* w4 :** $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ — uniquement 4-voisins.
- **Saturation :** $\text{clip}(x) = \max(0, \min(255, x))$ appliqué au résultat du rehaussement.
- **Sans arrondi :** Le Laplacien n'utilise que des additions/soustractions d'entiers.

3.12.7.3 🧠 Fondement Théorique

Région	$\nabla^2 f$	Effet du Rehaussement
Uniforme	≈ 0	Aucune modification
Contour croissant	< 0	Pixel éclairci
Contour décroissant	> 0	Pixel assombri

3.12.7.4 📦 Spécification d'Entrée et de Sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.

- Lignes suivantes : Éléments de la matrice originale.

Sortie :

- Matrice rehaussée $L \times C$.

3.12.7.5 📌 Exemples

Entrée	Sortie	Observation
3 3 0 0 0 0 100 0 0 0 0	0 0 0 0 255 0 0 0 0	Pic isolé : 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	Région uniforme : Laplacien=0, aucune modification

```
%%writefile EP03_07.py
# Code Python
```

```
Overwriting EP03_07.py
```

```
TestSuite("EP03_07.py").run()
```

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

3.12.8 EP03_08 🧭 Gradient de Sobel : Gx et Gy

Dans les robots explorateurs de Mars (comme Perseverance), la détection d'obstacles est réalisée en temps réel par des caméras stéréoscopiques. L'**opérateur de Sobel** calcule le gradient directionnel de la scène et est utilisé dans l'algorithme de détection de contours pour identifier les roches, les fissures et les dénivelés du terrain qui pourraient compromettre la navigation.

Voir dans Figure 3.33 une simulation de cet EP.

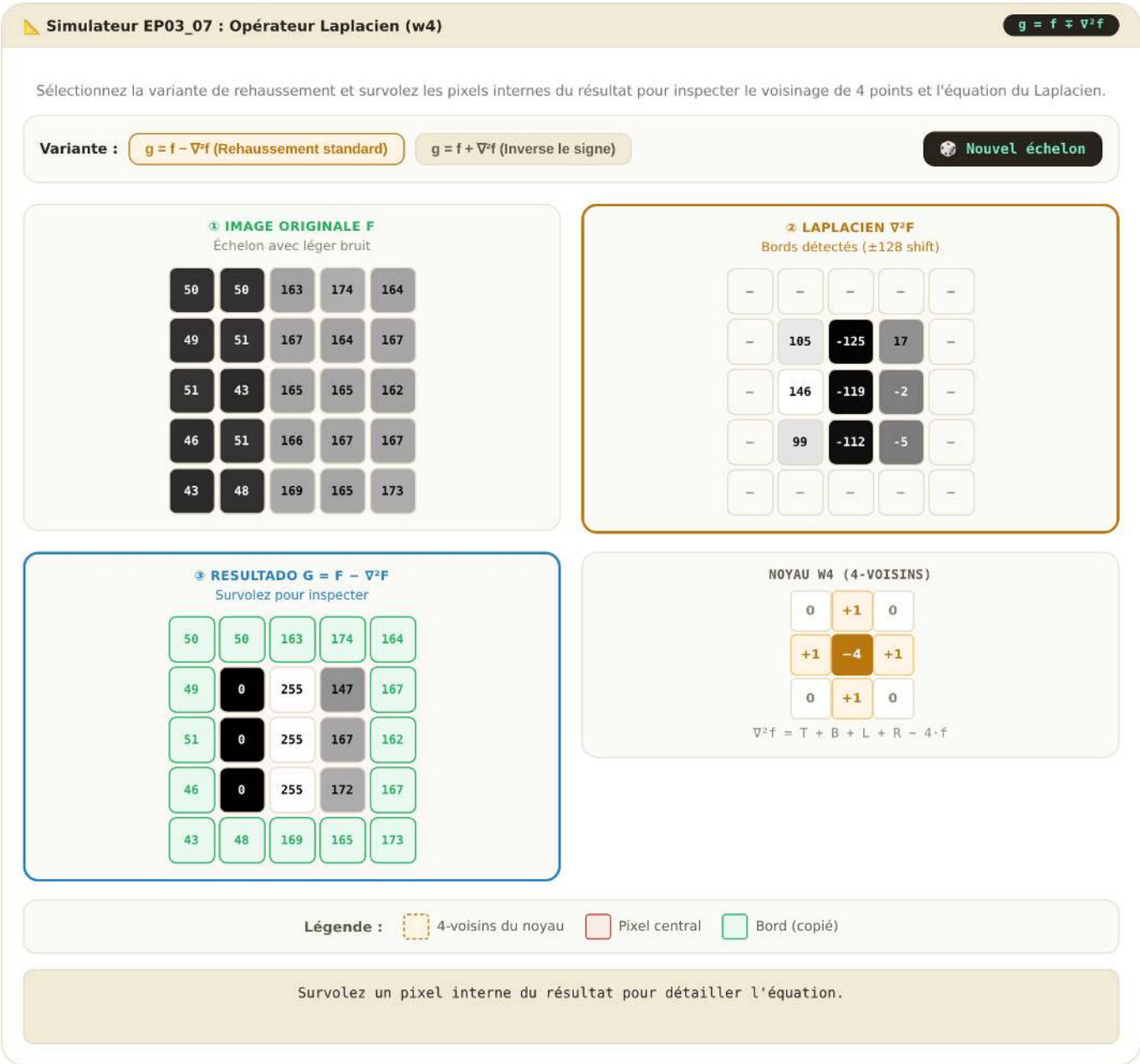
3.12.8.1 📋 Directives d'implémentation

1. **Dimensions :** Lire les entiers L (lignes) et C (colonnes).
2. **Données :** Lire la matrice f.
3. **Gx et Gy :** Pour chaque pixel **interne** (i, j) avec $1 \leq i < L-1$, $1 \leq j < C-1$:

$$G_x(i,j) = [f(i-1,j+1) + 2f(i,j+1) + f(i+1,j+1)] - [f(i-1,j-1) + 2f(i,j-1) + f(i+1,j-1)]$$

$$G_y(i,j) = [f(i+1,j-1) + 2f(i+1,j) + f(i+1,j+1)] - [f(i-1,j-1) + 2f(i-1,j) + f(i-1,j+1)]$$

4. **Magnitude :** $|\nabla f(i,j)| = \text{clip}(\text{round}(\sqrt{G_x^2 + G_y^2}))$.
5. **Bordure :** Les pixels de bordure reçoivent une magnitude de 0.
6. **Sortie :** Afficher la magnitude $L \times C$.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap03/sim-ep0307-laplaciano.html

Figure 3.32: Simulateur EP03_07 : Opérateur Laplacien (w4) pour le rehaussement des contours

3.12.8.2 📌 Contraintes computationnelles

- **Arrondi :** Appliquer `round` avant de convertir en entier.
- **Saturation :** $\text{clip}(x) = \max(0, \min(255, x))$.
- **Racine carrée :** Utiliser $\sqrt{G_x^2 + G_y^2}$ (pas l'approximation $|G_x| + |G_y|$).

3.12.8.3 🧠 Fondement théorique

Opérateur	Détecte	Coefficients diagonaux
G_x	Contours verticaux	± 1
G_y	Contours horizontaux	± 1
$\|\nabla f\|$	Tous les contours	Combiné

3.12.8.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Lignes suivantes : Éléments de la matrice.

Sortie :

- Magnitude du gradient, matrice $L \times C$.

3.12.8.5 📌 Exemples

Entrée	Sortie	Observation
3 3 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0	Image nulle : gradient zéro
3 3 0 0 255 0 0 255 0 0 255	0 0 0 0 255 0 0 0 0	Contour vertical central : Gx élevé

```
%%writefile EP03_08.py
# Code Python
```

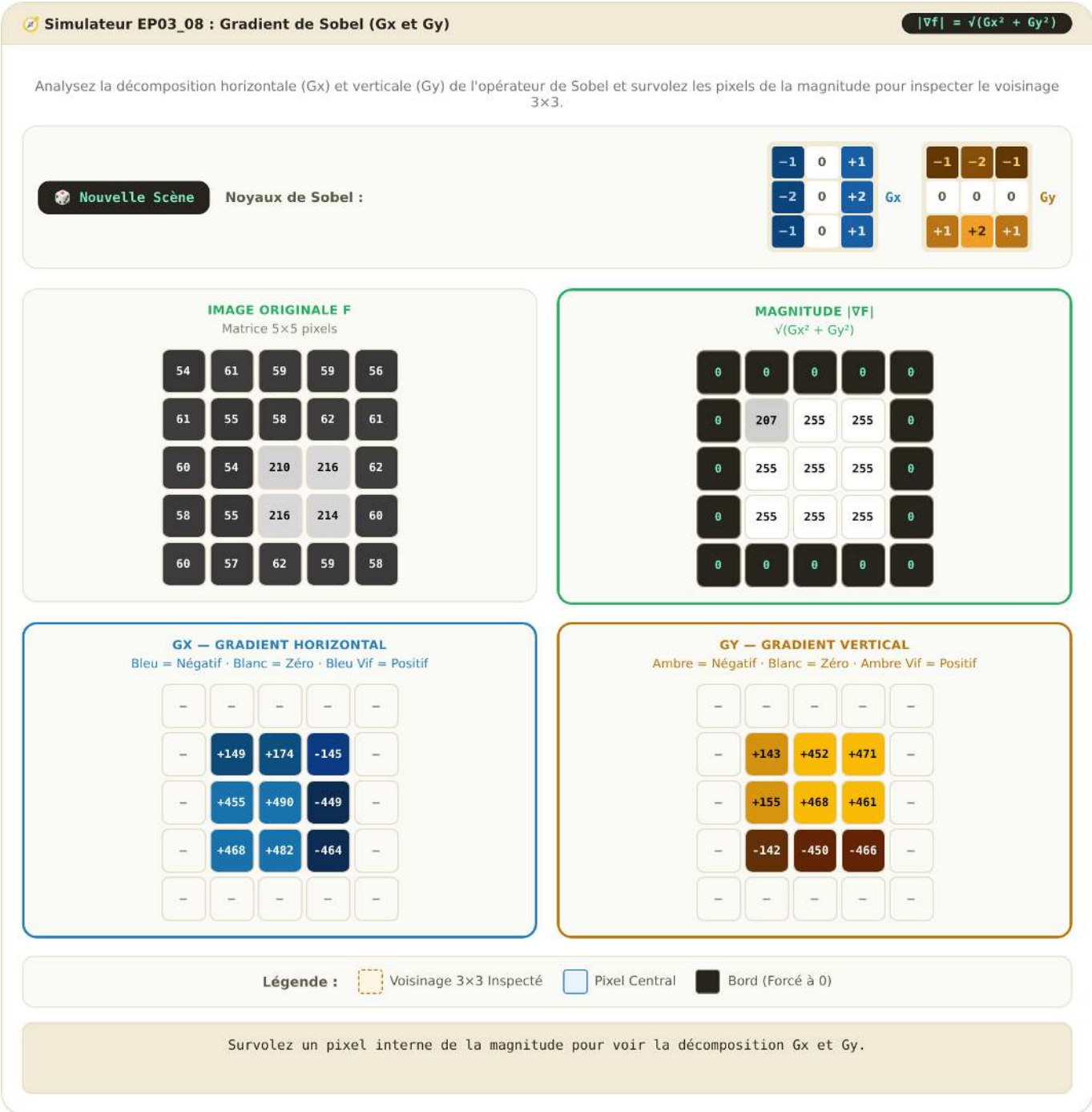
```
Overwriting EP03_08.py
```

```
TestSuite("EP03_08.py").run()
```

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

3.12.9 EP03_09 📡 Filtre Médian 3×3

Les images radar à synthèse d'ouverture (SAR) utilisées en surveillance environnementale et militaire souffrent d'un type spécifique de bruit appelé *speckle*, qui présente des caractéristiques similaires au bruit sel et poivre.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap03/sim-ep0308-sobel.html

Figure 3.33: Simulateur EP03_08 : Gradient de Sobel (Gx et Gy)

Le **filtre médian** est la méthode standard pour supprimer ce bruit, car il préserve les contours des structures tout en éliminant les points parasites.

Voir dans Figure 3.34 une simulation de cet EP.

3.12.9.1 📋 Directives d'implémentation

1. **Dimensions :** Lire les entiers L (lignes) et C (colonnes).
2. **Données :** Lire la matrice de pixels f.
3. **Filtre Médian 3×3 :** Pour chaque pixel **interne** (i, j) avec $1 \leq i < L-1$, $1 \leq j < C-1$:
   - Collecter les 9 pixels du voisinage 3×3 : $\{f(i+s, j+t) : s, t \in \{-1, 0, 1\}\}$.
   - Trier les 9 valeurs en ordre croissant.
   - Affecter $g(i, j)$ à la valeur centrale (position d'indice 4, en considérant l'indice 0).

$$g(i, j) = \text{médiane}\{f(i+s, j+t) : s, t \in \{-1, 0, 1\}\}$$

4. **Bordure :** Copier directement : $g(i, j) = f(i, j)$.
5. **Sortie :** Afficher la matrice filtrée $L \times C$.

3.12.9.2 📌 Contraintes computationnelles

- **Fenêtre :** Toujours $3 \times 3 = 9$ éléments.
- **Médiane :** L'élément central de la séquence triée (indice 4 de 0 à 8).
- **Pas de clipping :** La médiane des valeurs dans $[0, 255]$ reste dans $[0, 255]$.
- **Non linéaire :** Le filtre médian ne peut pas être exprimé comme une convolution linéaire.

3.12.9.3 🧠 Fondement théorique

Bruit	Filtre de moyenne	Filtre médian
Sel et poivre (0 ou 255)	Étale le bruit	Supprime sans déformer les contours
Gaussien	Réduit efficacement	Réduit partiellement

3.12.9.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Lignes suivantes : Éléments de la matrice.

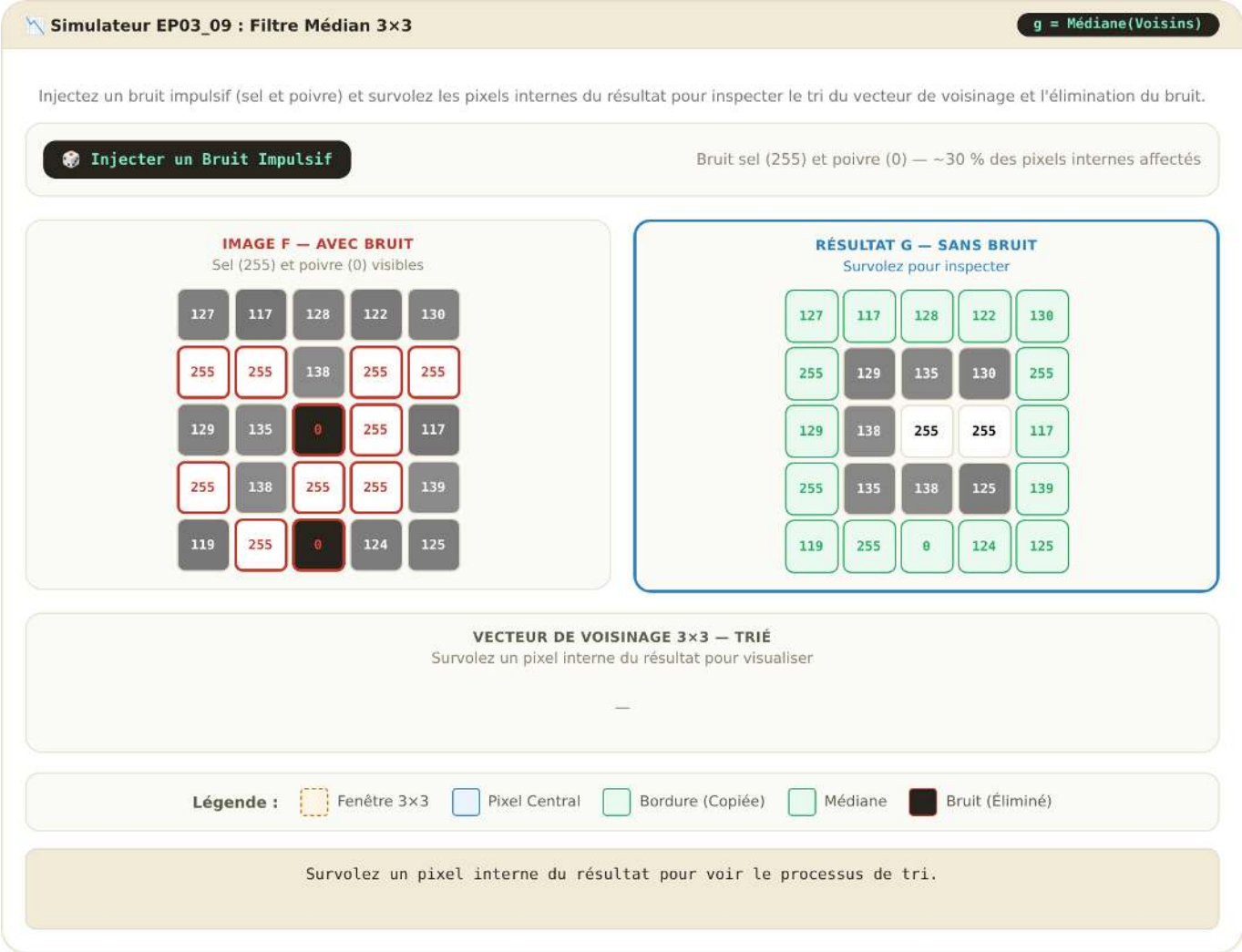
Sortie :

- Matrice filtrée $L \times C$.

3.12.9.5 📌 Exemples

Entrée	Sortie	Observation
3 3 100 100 100 100 0 100 100 100 100	100 100 100 100 100 100 100 100 100	Point noir éliminé : médiane de 8×100+1×0 = 100

Entrée	Sortie	Observation
3 3 50 50 50 50 255 50 50 50 50	50 50 50 50 50 50 50 50 50	Point blanc (sel) éliminé



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap03/sim-ep0309-mediana.html

Figure 3.34: Simulateur EP03_09 : Filtre de la Médiane 3×3

```
%%writefile EP03_09.py
# Code Python
```

```
Overwriting EP03_09.py
```

```
TestSuite("EP03_09.py").run()
```

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

3.12.10 EP03_10 ✨ *Unsharp Masking* (USM)

Dans les systèmes de numérisation de documents historiques et d'œuvres d'art, la netteté des images est essentielle pour la lecture de textes manuscrits et de détails ornementaux. L'***Unsharp Masking*** **(USM)**

est l'algorithme de rehaussement de netteté standard utilisé dans les *scanners* professionnels et les logiciels tels qu'Adobe Photoshop, contrôlé par le paramètre k qui détermine l'intensité du rehaussement.

Voir dans Figure 3.35 une simulation de cet EP.

3.12.10.1 📋 Directives d'implémentation

1. **Dimensions :** Lire les entiers L (lignes) et C (colonnes).
2. **Paramètre :** Lire la valeur réelle k (intensité du rehaussement, $k \geq 0$).
3. **Données :** Lire la matrice de pixels f.
4. **Lissage :** Calculer $\bar{f}$ avec un filtre de moyenne 3×3 (uniquement les pixels internes ; les bords sont conservés) :

$$\bar{f}(i,j) = \frac{1}{9} \sum_{s=-1}^{1} \sum_{t=-1}^{1} f(i+s, j+t)$$

5. **Masque haute fréquence :** $m(i,j) = f(i,j) - \bar{f}(i,j)$.
6. **Rehaussement USM :** Pour chaque pixel interne :

$$g(i,j) = \text{clip}\left(\text{round}\left(f(i,j) + k \cdot m(i,j)\right)\right)$$

7. **Bord :** $g(i,j) = f(i,j)$ (copie directe).
8. **Sortie :** Afficher la matrice rehaussée $L \times C$.

3.12.10.2 📌 Contraintes computationnelles

- **Arrondi :** Appliquer `round` avant le clipping.
- **Saturation :** $\text{clip}(x) = \max(0, \min(255, x))$.
- **Opérations en float :** Calculer $\bar{f}$ et m en virgule flottante avant d'arrondir le résultat final.
- $k = 0$ **:** Pas de rehaussement — la sortie est identique à l'entrée (sauf pour les bords).

3.12.10.3 🧠 Fondement théorique

Étape	Opération	Description
1	$\bar{f} = f * \frac{1}{9}\mathbf{1}_{3\times3}$	Lissage (basses fréquences)
2	$m = f - \bar{f}$	Masque (hautes fréquences)
3	$g = \text{clip}(\text{round}(f + k \cdot m))$	Rehaussement pondéré

3.12.10.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

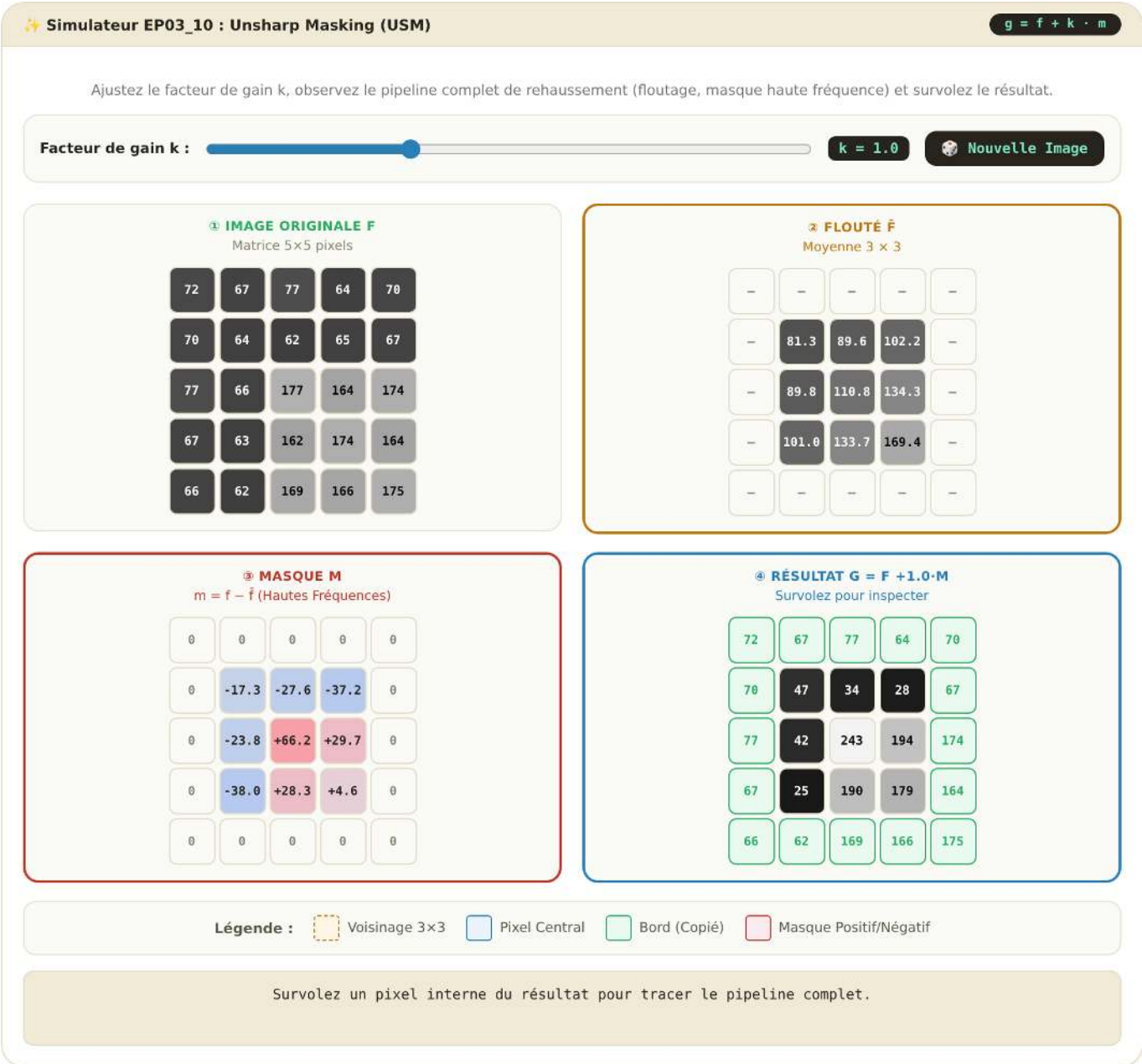
- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Réel k.
- Lignes suivantes : Éléments de la matrice originale.

Sortie :

- Matrice rehaussée $L \times C$.

3.12.10.5 📌 Exemples

Entrée	Sortie	Observation
3	100 100 100	k=0 : pas de rehaussement
3	100 100 100	
0.0	100 100 100	
100 100 100		
100 100 100		
100 100 100		
3	50 50 50	k=1 : pixel central rehaussé et saturé
3	50 255 50	
1.0	50 50 50	
50 50 50		
50 200 50		
50 50 50		



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap03/sim-ep0310-unsharp.html

Figure 3.35: Simulateur EP03_10 : *Unsharp Masking* (USM)

```
%%writefile EP03_10.py
# Code Python
```

```
Overwriting EP03_10.py
```

```
TestSuite("EP03_10.py").run()
```

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

Chapitre 4

Morphologie mathématique et segmentation d'images

Exécuter Colab Ouvrir GitHub

Ce chapitre présente deux thèmes fondamentaux du traitement numérique des images (TNI) : la **morphologie mathématique** et la **segmentation d'images**. La morphologie mathématique fournit un cadre théorique fondé sur la théorie des ensembles pour analyser, affiner et quantifier la forme des objets dans les images binaires et en niveaux de gris, au moyen d'opérateurs fondamentaux tels que l'**érosion** et la **dilatation**. La segmentation, quant à elle, vise à partitionner l'image en régions d'intérêt, en séparant les objets du fond et en produisant des représentations adaptées à l'analyse et à l'interprétation.

Le chapitre débute par le **seuillage**, l'une des techniques de segmentation les plus importantes, en introduisant la méthode automatique d'**Otsu** et en revisit l'analyse des histogrammes au moyen de la variance interclasses, présentée au chapitre 1. Ensuite, sont étudiés les principaux opérateurs de la morphologie mathématique, notamment l'érosion, la dilatation, l'ouverture, la fermeture et la reconstruction morphologique, qui permettent d'affiner les masques binaires et de préserver les structures pertinentes des objets. Enfin, sont présentées des techniques de segmentation fondée sur les régions, telles que l'**étiquetage des composantes connexes**, la **transformée de distance** et l'algorithme ***watershed*** basé sur les marqueurs, aboutissant à l'extraction de descripteurs géométriques et à la génération de *bounding boxes* compatibles avec les systèmes modernes de détection d'objets.

4.1 Objectifs

À la fin de ce chapitre, vous serez capable de :

- **Appliquer le seuillage :** Comprendre le critère automatique d'Otsu par maximisation de la variance interclasse (σ_B^2) et sélectionner des stratégies de prétraitement appropriées pour faciliter la segmentation ;
- **Maîtriser la morphologie binaire :** Comprendre et appliquer l'érosion ($A \ominus B$) et la dilatation ($A \oplus B$) comme opérateurs fondamentaux, en dérivant l'ouverture ($A \circ B$), la fermeture ($A \bullet B$) et les opérations basées sur la reconstruction morphologique, comme `mm.clohole` et `mm.edgeoff` ;
- **Appliquer la morphologie en niveaux de gris :** Utiliser le gradient morphologique et les filtres *top-hat* pour le rehaussement et l'analyse de structures locales ;
- **Étiqueter les composantes connexes :** Identifier et séparer les régions connectées dans des images binaires à l'aide d'algorithmes d'étiquetage ;
- **Appliquer la transformée de distance :** Interpréter et calculer les distances au fond en utilisant des approches morphologiques et des métriques géométriques ;
- **Segmenter par régions :** Construire des *pipelines* de segmentation basés sur des marqueurs utilisant la transformée de distance et l'algorithme ***watershed*** ;
- **Extraire des descripteurs géométriques :** Calculer des propriétés telles que l'aire, le périmètre, le centroïde, la circularité et les *bounding boxes* à l'aide de `mm.label0` et de l'extraction de contours ;
- **Relier le TNI et la vision par ordinateur :** Comprendre comment les descripteurs extraits par segmentation peuvent être convertis en formats utilisés par les détecteurs modernes, comme 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
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0
```

4.2 Seuilletage

Le **seuilletage** (*thresholding*) est l'une des formes les plus simples et efficaces de segmentation d'images. Son objectif est de classer chaque pixel en deux classes d'intensité, généralement associées à *objet* et *fond* :

$$g(x,y) = \begin{cases} 255, & \text{si } f(x,y) > T \\ 0, & \text{sinon} \end{cases} \tag{4.1}$$

où $f(x,y)$ représente l'intensité du pixel dans l'image originale et $g(x,y)$ l'image binaire résultante.

Le choix du seuil T est important pour la qualité de la segmentation. La méthode d'**Otsu** détermine automatiquement le seuil optimal en maximisant la **variance interclasses** σ_B^2 définie par :

$$\sigma_B^2(T) = w_0(T)\, w_1(T)\, \left[\mu_0(T) - \mu_1(T)\right]^2 \tag{4.2}$$

où :

- $w_0(T)$ et $w_1(T)$ sont les probabilités cumulées des classes fond et objet ;
- $\mu_0(T)$ et $\mu_1(T)$ sont les moyennes d'intensité de ces classes ;
- $\sigma_B^2(T)$ représente la variance interclasses pour un seuil donné T.

La méthode fonctionne mieux lorsque l'histogramme présente deux groupes d'intensités relativement séparés. Pour cela, l'algorithme évalue tous les seuils possibles de l'image — typiquement dans l'intervalle $[0, 255]$ pour des images 8 bits — et sélectionne la valeur qui maximise la variance entre classes, notée σ_B^2 :

$$T^* = \arg \max_{T \in [0,255]} \sigma_B^2(T)$$

ℹ Otsu suppose des histogrammes bimodaux

La méthode d'Otsu produit de meilleurs résultats lorsque l'histogramme présente deux pics bien définis (*bimodalité*), correspondant au fond et à l'objet. Plus la séparation entre ces pics est grande et plus le maximum de σ_B^2 est prononcé, plus le seuil obtenu tend à être fiable.
Dans les images avec un éclairage non uniforme ou de multiples régions d'intensité, les techniques de **seuilletage adaptatif** — dans lesquelles le seuil est calculé localement — produisent généralement des segmentations plus robustes.
L'indice B dans σ_B^2 signifie ***between classes*** (*entre classes*). Ainsi, σ_B^2 représente la **variance entre les classes** (*between-class variance*).

4.2.1 Image de pièces de monnaie

L'image utilisée pour pratiquer la segmentation est une photographie d'une collection de pièces de monnaie de différents pays et époques (Figure 4.1). Crédit : GAZI.MD.AHAD (CC BY-SA 4.0). Elle présente des objets circulaires aux contours bien définis, ce qui la rend idéale pour illustrer le seuillage, les opérateurs morphologiques, la transformée de distance, le *watershed* et les descripteurs de forme.

```
import os

url     = "https://upload.wikimedia.org/wikipedia/commons/2/25/GAZI.MD.AHAD_11.jpg"
caminho = "imagens/coins.png"

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

img_coins_color = mm.read(caminho)
img_coins_gray  = mm.gray(img_coins_color)

print(f"Dimensions [y,x,c] : {img_coins_color.shape}")
mm.show(img_coins_color, scale=30)
```

```
Dimensions [y,x,c] : (2560, 1920, 3)
```

Figure 4.1: Image avec des pièces de plusieurs types. Crédit : GAZI.MD.AHAD (CC BY-SA 4.0).

4.2.2 Prétraitement pour Otsu

La qualité de la méthode d'Otsu dépend directement du degré de **bimodalité** de l'histogramme de l'image d'entrée. L'image originale des pièces présente un éclairage non uniforme et des pièces sombres (cuivre oxydé) avec des intensités proches de celles du fond sombre, rendant l'histogramme peu bimodal.

Pour minimiser ces limitations, deux techniques classiques de prétraitement, présentées dans le chapitre précédent, seront évaluées et appliquées avant l'étape de seuillage. Table 4.1 résume les caractéristiques de

chaque approche. Ces techniques visent à augmenter la séparation entre l'objet et le fond, rapprochant ainsi l'histogramme d'une distribution bimodale.

Tableau 4.1: Techniques de prétraitement évaluées pour améliorer la séparation entre les pièces et le fond avant l'application de la méthode d'Otsu.

Technique	Ce qu'elle fait	Quand l'utiliser
CLAHE	Égalisation d'histogramme adaptative locale	Faible contraste global ou régional
Gaussien	Lissage par convolution avec une gaussienne	Bruit haute fréquence (texture du fond)

La fonction `cv2.createCLAHE(clipLimit, tileGridSize)` divise l'image en blocs et applique une égalisation d'histogramme à chacun d'eux, en limitant l'amplification du bruit par le biais du paramètre `clipLimit`. Quant au filtre Gaussien (`cv2.GaussianBlur`), il lisse les textures fines qui pourraient créer de faux pics dans l'histogramme.

Pour comparer objectivement quel prétraitement produit la meilleure entrée pour la méthode d'Otsu, seront présentés, pour chaque version de l'image, l'image elle-même, l'histogramme avec le seuil optimal T^* mis en évidence, la courbe $\sigma_B^2(T)$ et le résultat de la binarisation.

Critère de comparaison

La version qui présente la plus grande valeur de σ_B^2 à son pic fournit la meilleure séparation entre les classes de fond et d'objet et, par conséquent, la meilleure entrée pour la méthode d'Otsu (Figure 4.2).

```
import cv2, io
import matplotlib.pyplot as plt

def otsu_criterio(img):
    h=mm.hist(img); p=h/h.sum();                  # histogramme et probabilités
    sigma2=np.zeros(len(p))
    for T in range(1,len(p)):                     # parcourt les seuils
        w0,w1=p[:T].sum(),p[T:].sum()             # probabilités des classes
        if w0*w1==0: continue                     # évite la division par zéro
        mu0=(np.arange(T)*p[:T]).sum()/w0         # moyenne fond
        mu1=(np.arange(T,len(p))*p[T:]).sum()/w1  # moyenne objet
        sigma2[T]=w0*w1*(mu0-mu1)**2              #  ²B(T)
    return sigma2,np.argmax(sigma2)               # courbe et T optimal

def fig2img(fig):
    b=io.BytesIO(); fig.savefig(b,format='png',dpi=100)   # figure → tampon
    plt.close(fig); b.seek(0)
    return np.array(plt.imread(b))                        # tampon → tableau

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}")       # seuil optimal
    ax.set(xlabel="T" if ylabel=="  ²B" else "Intensidade",ylabel=ylabel)
    ax.legend(fontsize=8); plt.tight_layout()
    return fig2img(fig)

#   Prétraitements
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)]

#   Tableau comparatif
print(f"{'Versão':<18}{'T*':>6}{' ²B pico':>14}")
print("-"*40)

imgs,titles=[],[]
for nome,img in imgs0:
    sigma2,T=otsu_criterio(img)                          # calcule ²B(T)
    print(f"{nome:<18}{T:>6}{sigma2[T]:>14.4e}")
    imgs += [
        img,                                             # image
        plot_curve(mm.hist(img),T,f"Hist T*={T}","Freq.","steelblue"),
        plot_curve(sigma2,T," ²B(T)"," ²B","darkorange"),
        mm.threshold(img)                               # Otsu final
    ]
    titles += [nome,f"Hist T*={T}"," ²B(T)",f"Otsu T*={T}"]

#   Affichage 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
```

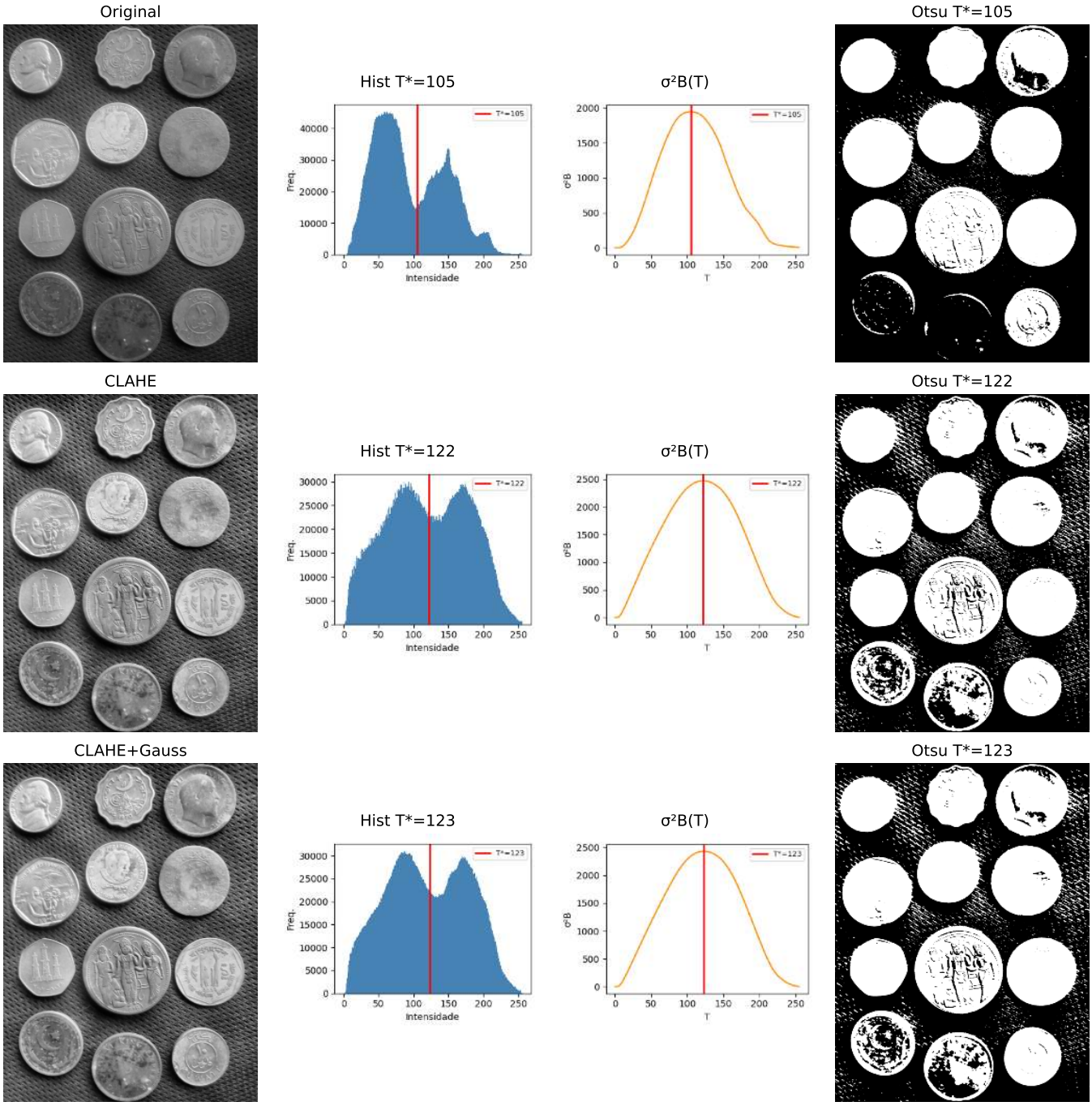


Figure 4.2: Comparaison des prétraitements : image | histogramme+T* | ²B(T) | Otsu. La meilleure séparation bimodale indique le seuil le plus fiable.

4.2.3 Résultat : CLAHE comme meilleur prétraitement

L'analyse de la Figure 4.2 indique que le **CLAHE** a obtenu la valeur la plus élevée de variance inter-classes ($\sigma_B^2 \approx 2{,}47 \times 10^3$), avec un seuil optimal $T^* = 122$. Bien que la combinaison **CLAHE+Gaussien** ait produit un résultat très similaire ($\sigma_B^2 \approx 2{,}43 \times 10^3$, $T^* = 123$), le critère quantitatif de la méthode d'Otsu favorise légèrement l'utilisation du CLAHE seul.

Sur le plan visuel, les images binarisées obtenues avec le CLAHE et le CLAHE+Gaussien sont pratiquement équivalentes. La différence entre les deux approches devient plus évidente dans l'analyse des histogrammes et des valeurs de $\sigma_B^2(T)$ que dans l'inspection directe des segmentations résultantes. Ainsi, le choix du CLAHE repose principalement sur la maximisation de la séparation statistique entre les classes de fond et d'objet.

Interprétation des résultats

Remarquez que les prétraitements avec CLAHE et CLAHE+Gaussien produisent des histogrammes et des seuils optimaux très proches ($T^* = 122$ et $T^* = 123$). Par conséquent, les images binarisées résultantes sont également très similaires. Dans ce cas, la décision ne repose pas sur des différences visuelles marquantes, mais sur le critère objectif de la méthode d'Otsu : la valeur la plus élevée de σ_B^2 indique la meilleure séparation entre les classes.

4.3 Morphologie mathématique

La **morphologie mathématique** est une théorie basée sur les ensembles, utilisée pour analyser la forme et la structure des objets dans les images. Contrairement aux filtres linéaires présentés au chapitre 3, les opérateurs morphologiques sont **non linéaires**, car ils reposent sur des opérations de minimum, de maximum et d'inclusion spatiale, plutôt que sur des combinaisons linéaires d'intensités. Ces opérateurs agissent sur le voisinage de chaque pixel au moyen d'un **élément structurant** $\mathbb{B}$, chargé de définir la forme et la taille de la région analysée.

Dans les images binaires et en niveaux de gris avec des éléments plans, l'élément structurant translaté à la position x est défini spatialement comme :

$$\mathbb{B}_x = \{x + b \mid b \in \mathbb{B}\}$$

Dans les régions de bord de l'image, une partie de l'ensemble $\mathbb{B}_x$ peut dépasser le domaine physique de la scène ($\mathbb{E}$). Pour garantir la cohérence mathématique des opérateurs primitifs à ces frontières, on suppose théoriquement que l'espace extérieur au domaine de l'image est rempli avec l'**élément neutre** de l'opération correspondante (infini positif pour l'érosion et infini négatif pour la dilatation), empêchant ainsi que l'environnement externe corrompe les structures internes de l'objet.

Lorsque l'élément structurant associe des poids à ses éléments — c'est-à-dire $b : \mathbb{B} \to \mathbb{Z}$ — il est appelé **fonction structurante** ou **élément structurant non plan**.

Développée par Matheron et Serra dans les années 1960 pour les images binaires, puis étendue aux niveaux de gris, la morphologie mathématique fonde des opérateurs tels que le gradient morphologique, le *top-hat*, le *watershed* et la transformée de distance, tous dérivés de deux primitifs : l'**érosion** et la **dilatation** [Matheron (1975); Serra (1982)].

4.3.1 Érosion et Dilatation

Les deux opérateurs primitifs sont définis de manière unifiée pour les images en niveaux de gris ($f : \mathbb{E} \to \mathbb{Z}$) et, par restriction au domaine $\{0, 1\}$, également pour les images binaires.

4.3.1.1 Érosion

L'**érosion** d'une image f par un élément structurant $b : \mathbb{B} \to \mathbb{Z}$ est définie formellement par :

$$\varepsilon_b(f)(x) = (f \ominus b)(x) = \min_{z \in \mathbb{B}} \{ f(x + z) - b(z) \}, \quad \forall\, x \in \mathbb{E} \tag{4.3}$$

En pratique, l'érosion remplace l'intensité du pixel x par la valeur minimale résultant de la différence entre l'image et l'élément structurant dans le voisinage défini par le domaine $\mathbb{B}$. Des valeurs positives dans les poids de $b(z)$ forcent le résultat local vers le bas, « creusant » plus profondément le relief de l'image et intensifiant l'érosion.

Dans le cas **plat** (où les poids sont nuls à l'intérieur du domaine, c'est-à-dire $b \equiv 0$), l'expression se simplifie en le minimum local pur :

$$\varepsilon_B(f)(x) = \min\{\, f(y) : y \in \mathbb{B}_x \,\}$$

Dans les images binaires, cette opération équivaut à exiger que l'ensemble $\mathbb{B}$, translaté à la coordonnée x, soit **complètement contenu** dans l'objet A :

$$A \ominus \mathbb{B} = \{\, z \in \mathbb{E} \mid \mathbb{B}_z \subseteq A \,\}$$

Effet visuel : *Rétrécit* les objets et les structures claires, éliminant les protubérances, les pics brillants ou les bruits qui sont géométriquement plus petits que le domaine $\mathbb{B}$.

4.3.1.2 Implémentation de l'érosion

La version didactique `mm.ero0` implémente le cas particulier de l'érosion avec **élément structurant plat**. Pour chaque pixel (y, x), la fonction parcourt les voisins spatiaux autorisés par B et stocke la plus petite valeur trouvée dans l'image d'entrée f, reproduisant directement l'opération de minimum local décrite dans la Équation 4.3 pour $b \equiv 0$.

Notez que les valeurs des voisins sont toujours lues de manière statique depuis l'image originale f ; la matrice de sortie g est utilisée exclusivement pour enregistrer le minimum accumulé du voisinage courant. Ainsi, le résultat final est invariant par rapport à l'ordre de balayage des pixels (que ce soit par lignes ou par colonnes).

La fonction auxiliaire `_viz` calcule les coordonnées des voisins valides dans les limites physiques de l'image. Sur les bords, l'initialisation de l'accumulateur à `255` imite avec exactitude le remplissage par élément neutre exigé par la théorie. Quant à la fonction d'interface `mm.ero`, elle recourt à l'implémentation native et optimisée d'OpenCV (`mm.ero`) lorsque l'élément structurant est plat, basculant vers la routine générale `mm.ero1` si l'élément possède des poids topographiques.

L'exemple computationnel suivant illustre l'application d'un élément structurant en croix (`mm.secross()`) mis en évidence dans la Figure 4.3, en comparant l'exécution de la variante didactique en boucle (`mm.ero0`) avec le moteur computationnel d'OpenCV (`mm.ero`).

```
B_cruz = mm.secross()
mm.drawImgPlt(B_cruz, scale=40)
```

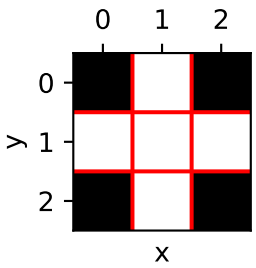


Figure 4.3: Élément structurant en forme de croix (B_{cruz}) utilisé pour la connectivité-4.

```
def _viz(f, B, y, x):
    """Génère (vy, vx, b_val) pour chaque voisin valide de (y,x)."""
    H, W = f.shape
    Bh, Bw = B.shape
    oh, ow = -Bh/2 + 0.5, -Bw/2 + 0.5  # offsets fixes
    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')):
    """Érosion (OpenCV ou avec poids)."""
    try:    return cv2.erode(f, Bc)
    except: return mm.ero1(f, Bc)

def ero0(f, Bc=np.ones((3,3),dtype='uint8')):
    """Érosion classique sans poids."""
    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 Test et de Validation
B = mm.secross()
print("Élément structurant B :")
print(mm.drawImage(B))

f = mm.randomImage(5,5)
print("Image originale f :")
print(mm.drawImage(f))

print("Érosion plane didactique (ero0) :")
print(mm.drawImage(ero0(f, B)))

print("Érosion optimisée OpenCV (ero) :")
print(mm.drawImage(ero(f, B)))
```

```
Élément structurant B :
0 1 0
1 1 1
0 1 0

Image originale f :
1 7 1 6 9
3 1 2 0 0
1 2 4 1 7
9 8 1 4 9
9 1 0 0 4

Érosion plane didactique (ero0) :
1 1 1 0 0
1 1 0 0 0
1 1 1 0 0
1 1 0 0 4
1 0 0 0 0

Érosion optimisée OpenCV (ero) :
1 1 1 0 0
1 1 0 0 0
1 1 1 0 0
1 1 0 0 4
1 0 0 0 0
```

La fonction `_viz` utilise `yield` pour générer chaque voisin à la demande, sans stocker tous les résultats en mémoire dans une liste. Dans l'expérience ci-dessous, une fenêtre de 3000×3000 produit 9 millions de voisins. L'implémentation basée sur une liste a consommé plus de 1 Go de RAM et a pris environ 46 s, tandis que la version avec `yield` a consommé une mémoire négligeable et s'est exécutée en 38 s. En traitement d'images, les générateurs sont particulièrement utiles pour parcourir de grandes voisinages de manière efficace.

```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} Mo ")
```

```
LISTA : 830.8 Mo
YIELD : 0.0 Mo
```

4.3.1.3 Dilatation

La **Dilatation** d'une image f par une fonction structurante $b : \mathbb{B} \to \mathbb{Z}$ est définie formellement par :

$$\delta_b(f)(x) = (f \oplus b)(x) = \max_{z \in \mathbb{B}} \{ f(x - z) + b(z) \}, \quad \forall\, x \in \mathbb{E} \tag{4.4}$$

En pratique, la dilatation remplace l'intensité du pixel x par la plus grande valeur résultant de la somme entre l'image et l'élément structurant dans le voisinage défini. L'argument d'inversion spatiale $(x - z)$ indique que la dilatation évalue implicitement l'élément transposé (réfléchi) $\hat{b}$, propriété fondamentale pour assurer la dualité mathématique par rapport à l'érosion.

Dans le cas **plat** (où les poids sont nuls à l'intérieur du domaine, c'est-à-dire $b \equiv 0$), l'expression se réduit au maximum local pur :

$$\delta_B(f)(x) = \max\{ f(y) : y \in \mathbb{B}_x \}$$

Dans les images binaires, cette opération équivaut à exiger que l'ensemble réfléchi $\hat{\mathbb{B}}$, translaté à la coordonnée x, possède une **intersection non vide** avec l'objet A :

$$A \oplus \mathbb{B} = \{ z \in \mathbb{E} \mid \hat{\mathbb{B}}_z \cap A \neq \emptyset \}$$

Effet visuel : *Étend* les structures claires de l'image, augmente le remplissage des objets, connecte les composantes proches et élimine les canaux, fosses sombres ou vallées qui sont géométriquement plus petits que le domaine $\mathbb{B}$.

4.3.1.4 Implémentation de la dilatation

La version didactique `mm.dil0` implémente le cas particulier de dilatation avec **élément structurant plan**. Pour chaque pixel (y, x), la fonction parcourt les voisins spatiaux autorisés par B et stocke la plus grande valeur trouvée dans l'image d'entrée f, reproduisant directement l'opération de maximum local pour $b \equiv 0$.

Avant de lancer le balayage spatial, l'élément structurant subit une réflexion géométrique au moyen de l'instruction `np.flip(Bc)` afin de construire explicitement la matrice transposée $\hat{B}$ exigée par la théorie. Pour des masques parfaitement symétriques (tels que des croix, des carrés et des disques centrés à l'origine), cette réflexion ne modifie pas l'arrangement des pixels ; toutefois, pour les éléments asymétriques, cette étape est strictement nécessaire afin de garantir l'équivalence avec les définitions formelles et de préserver les lois de dualité.

Comme constaté pour l'opérateur d'érosion, les valeurs des voisins sont toujours lues de manière statique à partir de la matrice originale f, tandis que la matrice de sortie g agit uniquement comme le registre du

maximum accumulé du voisinage. Aux frontières de l'image, l'initialisation de l'accumulateur à `0` émule avec exactitude le remplissage externe par élément neutre ($-\infty$, ou zéro dans les représentations sur 8 bits), garantissant que les bords physiques de la scène soient dilatés en parfaite conformité avec le standard adopté par OpenCV.

L'exemple informatique ci-dessous illustre l'application pratique d'un élément en croix (`mm.secross()`), validant la cohérence entre la logique en boucles (`mm.dil0`) et la méthode native industrielle (`mm.dil`).

```
def dil(f, Bc=np.zeros((3,3),dtype='uint8')):
    """Dilatation (OpenCV ou avec des poids)."""
    try:    return cv2.dilate(f, Bc)
    except: return mm.dil1(f, Bc)

def dil0(f, Bc=np.zeros((3,3),dtype='uint8')):
    """Dilatation plane suivant rigoureusement la théorie."""
    g = np.empty_like(f)
    Bc = np.flip(Bc)     # réflexion explicite : B
    for y in range(f.shape[0]):
        for x in range(f.shape[1]):
            g[y,x] = 0 # Initialise avec la valeur minimale pour chercher le maximum
            for vy,vx,bv in mm._viz(f,Bc,y,x):
                if bv and g[y,x] < f[vy,vx]:
                    g[y,x] = f[vy,vx]
    return g

# Script de test et de validation
B = mm.secross()
print("Élément structurant B :")
print(mm.drawImage(B))

f = mm.randomImage(5,5)
print("Image originale f :")
print(mm.drawImage(f))

print("Dilatation plane didactique (dil0) :")
print(mm.drawImage(dil0(f, B)))

print("Dilatation optimisée OpenCV (dil) :")
print(mm.drawImage(dil(f, B)))
```

```
Élément structurant B :
0 1 0
1 1 1
0 1 0

Image originale f :
7 4 0 7 7
8 9 2 4 1
4 5 7 5 5
8 7 2 6 2
6 3 0 3 1

Dilatation plane didactique (dil0) :
8 9 7 7 7
9 9 9 7 7
8 9 7 7 5
8 8 7 6 6
8 7 3 6 3

Dilatation optimisée OpenCV (dil) :
```

```
8 9 7 7 7
9 9 9 7 7
8 9 7 7 5
8 8 7 6 6
8 7 3 6 3
```

Note : Le Confront des Signes ($f(x+z)$ vs $f(x-z)$)

Comparez les définitions formelles de l'érosion (Équation 4.3) et de la dilatation (Équation 4.4). Considérez un élément structurant asymétrique à droite $\mathbb{B} = \{0, 1\}$ (origine et un pixel à droite) appliqué à la position $x = 10$.

1. **Dans l'érosion** (Équation 4.3) :

$$\min\{f(x+z) - b(z)\}$$

   - $z = 0 \Rightarrow f(10+0) = \mathbf{f(10)}$
   - $z = 1 \Rightarrow f(10+1) = \mathbf{f(11)}$

   L'opérateur consulte le pixel courant (10) et le pixel à droite (11), préservant l'orientation originale de $\mathbb{B}$.

2. **Dans la dilatation** (Équation 4.4) :

$$\max\{f(x-z) + b(z)\}$$

   - $z = 0 \Rightarrow f(10-0) = \mathbf{f(10)}$
   - $z = 1 \Rightarrow f(10-1) = \mathbf{f(9)}$

   En raison du signe négatif ($-z$), avancer dans l'élément structurant correspond à reculer dans l'image, ce qui fait que la dilatation consulte le pixel à gauche (9).

La fonction `_viz`, utilisée dans `morph.py`, génère des voisins par déplacements additifs de la forme $x+z$. Pour cette raison, l'implémentation de `mm.dil0` reflète au préalable l'élément structurant au moyen de `np.flip(B)`. Après la réflexion, le balayage basé sur $x+z$ accède alors exactement aux mêmes points définis par l'expression théorique $f(x-z)$ de la dilatation dans Équation 4.4.
Pour les éléments structurants symétriques (comme les disques, les carrés et les croix centrées), la réflexion ne modifie pas le masque. En revanche, pour les éléments asymétriques, cette étape est indispensable pour que l'implémentation reproduise correctement la définition mathématique de la dilatation et préserve la dualité érosion–dilatation.

Dualité érosion–dilatation

L'érosion et la dilatation sont **duales par complémentation**. Cela signifie qu'un opérateur peut être entièrement obtenu à partir de l'autre, à condition d'agir sur le complément de l'image en utilisant l'élément structurant réfléchi $\hat{B}$:

$$(A \ominus B)^c = A^c \oplus \hat{B} \quad \Longleftrightarrow \quad A \ominus B = (A^c \oplus \hat{B})^c$$

De manière analogue, la **dilatation peut également être obtenue à partir de l'érosion** :

$$(A \oplus B)^c = A^c \ominus \hat{B} \quad \Longleftrightarrow \quad A \oplus B = (A^c \ominus \hat{B})^c$$

En termes pratiques, l'érosion d'un objet peut être obtenue par la dilatation de son complément, suivie de la complémentation du résultat (et vice-versa). Dans l'implémentation du paquet `morph.py`, les versions didactiques `mm.ero0` et `mm.dil0` rendent explicite cette structure au moyen de boucles (*loops*), tandis que `mm.ero` et `mm.dil` délèguent les opérations à OpenCV pour une plus grande efficacité computationnelle.

Conditions aux limites et images finies

En morphologie mathématique classique, définie sur un domaine infini (typiquement $\mathbb{Z}^2$), cette dualité est exacte. Dans les images numériques, cependant, on travaille avec des matrices finies, et le résultat dépend de la manière dont sont traités les pixels situés hors de l'image.
Pour que les identités de dualité restent valides, le complément doit être défini par rapport au même univers et les conditions aux limites adoptées pour l'érosion et pour la dilatation doivent être complémentaires entre elles. Par exemple, si l'érosion suppose que les pixels externes appartiennent à l'objet (255), alors la dilatation appliquée au complément doit supposer que ces mêmes pixels externes appartiennent au fond (0).
Lorsque différentes stratégies de remplissage sont utilisées (réplication, réflexion, valeur constante, etc.), la dualité théorique peut cesser d'être satisfaite exactement dans les régions proches des bords de l'image.

Pour illustrer numériquement les opérateurs morphologiques et la dualité érosion–dilatation, la Figure 4.4 présente une image binaire 10×10 traitée avec un élément structurant en forme de « L ». Dans l'implémentation de `morph.py`, l'origine de B est fixée au centre géométrique du masque — position $(1, 1)$ pour un *kernel* 3×3 — et doit correspondre à un élément actif pour que l'érosion se comporte correctement (comme discuté précédemment). L'élément structurant B_L défini ci-dessous satisfait cette condition. La Figure 4.5 complète l'analyse avec un simulateur interactif de l'érosion, permettant de visualiser le déplacement de l'élément structurant sur l'image et d'identifier les positions où il reste complètement contenu dans l'objet.

```
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', origine au centre
B_hat = np.array([[0,1,1],[0,1,1],[0,0,1]], dtype=np.uint8)   # B_L tourné de 180° (B)

print("Élément structurant B_L :"); print(mm.drawImg(B_L))
print("Élément structurant réfléchi B :"); print(mm.drawImg(B_hat))

img_ero0 = mm.ero0(A, B_L)

# Dualité : (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"Dualité (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)
)
```

```
Élément structurant B_L :
1 0 0
1 1 0
```

```
1 1 0

Élément structurant réfléchi B :
0 1 1
0 1 1
0 0 1

Dualité (A   B)  == A   B : True
```

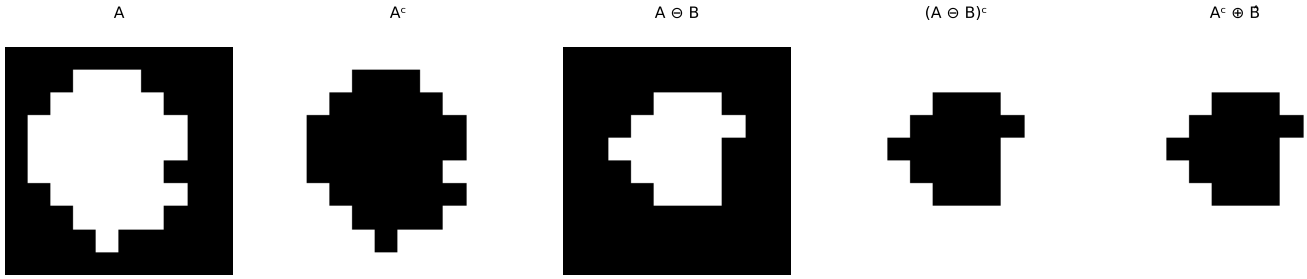
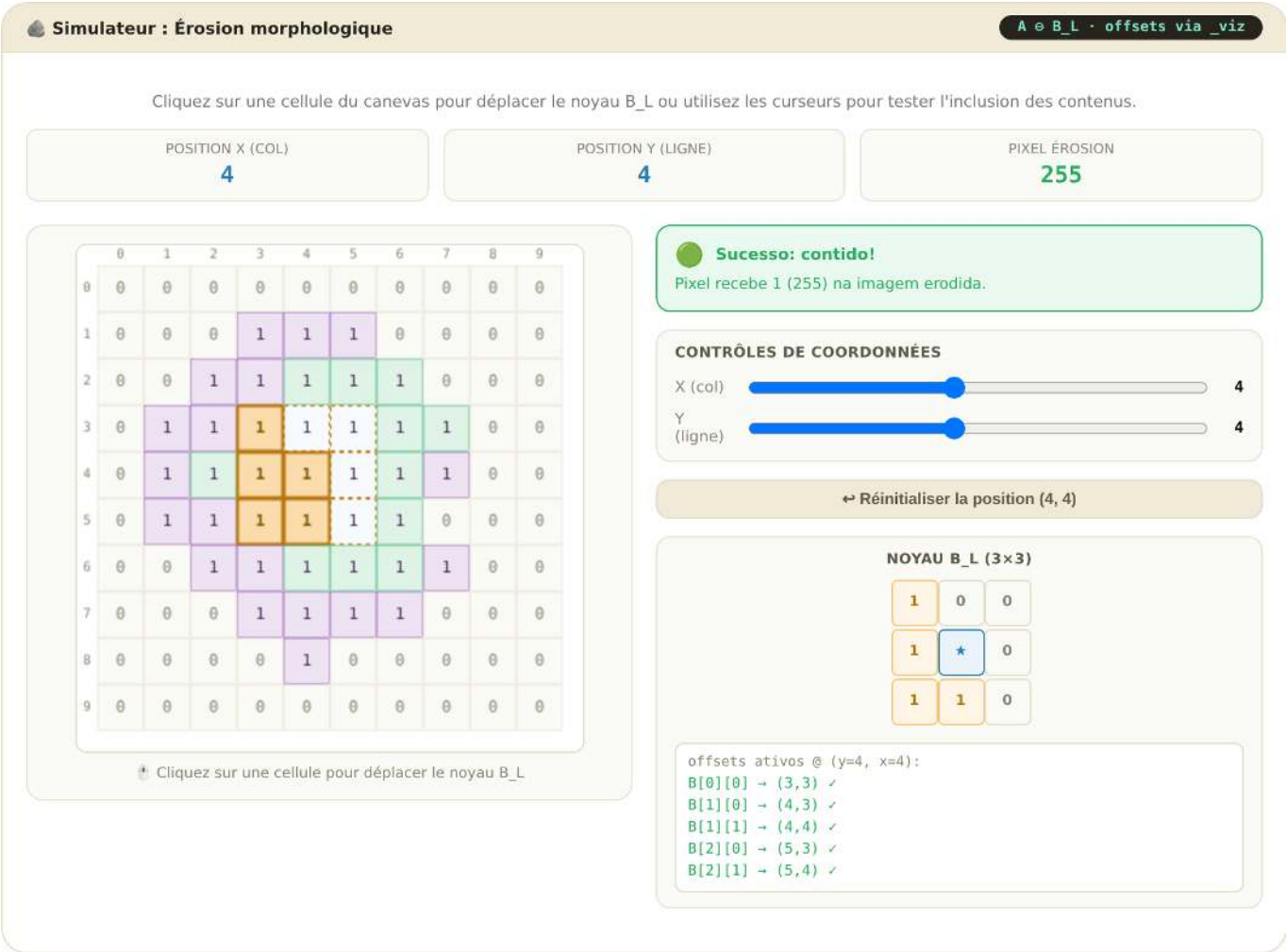


Figure 4.4: Érosion et dilatation sur une image binaire 10×10 avec un élément structurant « L » asymétrique 3×3. Validation de la dualité érosion–dilatation.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-04-erosao.html

Figure 4.5: Simulateur : Érosion Morphologique (A B_L)

4.3.2 Ouverture et Fermeture

En combinant érosion et dilatation, on obtient deux opérateurs d'une grande utilité pratique : l'**ouverture** et la **fermeture**, définis par les Équations Équation 4.5 et Équation 4.6. Leurs principaux effets sont résumés dans la Table 4.2.

Ouverture (*opening*) — érosion suivie d'une dilatation par le même B :

$$A \circ B = (A \ominus B) \oplus B \tag{4.5}$$

Fermeture (*closing*) — dilatation suivie d'une érosion par le même B :

$$A \bullet B = (A \oplus B) \ominus B \tag{4.6}$$

En pratique, `mm.open` et `mm.close` appliquent le même élément structurant aux deux étapes. Pour les éléments structurants symétriques (les plus courants), cette implémentation coïncide avec la définition mathématique présentée ci-dessus.

Tableau 4.2: Propriétés de l'ouverture et de la fermeture.

Opérateur	Séquence	Effet principal
Ouverture $A \circ B$	érosion → dilatation	Supprime les structures incapables de contenir l'élément structurant ; lisse les contours externes
Fermeture $A \bullet B$	dilatation → érosion	Comble les trous plus petits que B ; lisse les contours internes

Propriété importante : les deux sont **idempotents**. Par exemple,

$$(A \circ B) \circ B = A \circ B,$$

c'est-à-dire qu'après la première application, de nouvelles applications du même opérateur ne modifient plus le résultat.

4.3.2.1 Implémentation de l'ouverture et de la fermeture

Contrairement à l'érosion et à la dilatation, l'ouverture et la fermeture n'introduisent pas de nouveaux mécanismes computationnels. Les deux sont obtenus par la composition séquentielle des opérateurs primitifs déjà présentés :

```
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 fonction `mm.open` délègue l'opération à `mm.open(f, B)`, tandis que `mm.close` utilise `mm.close(f, B)`, produisant le même résultat de manière plus efficace.

L'ouverture hérite de l'érosion la capacité de supprimer les structures plus petites que l'élément structurant et de la dilatation la restauration partielle des régions préservées. La fermeture réalise le processus inverse : elle expand d'abord les objets puis restaure leurs dimensions originales, comblant les lacunes et les trous plus petits que l'élément structurant.

4.3.2.2 Filtre Séquentiel Alterné

En pratique, l'ouverture et la fermeture sont souvent appliquées en séquence afin d'éliminer simultanément le bruit externe et de combler les trous internes. La fonction `mm.asf` (*Alternating Sequential Filter*) généralise cette stratégie en appliquant alternativement des ouvertures et des fermetures avec des éléments structurants progressivement plus grands. Les séquences disponibles sont présentées dans Table 4.3.

Tableau 4.3: Séquences du filtre séquentiel alterné mm.asf.

Séquence	Ordre	Utilisation typique
`'OC'`	ouverture → fermeture	élimine le bruit externe avant de combler les petits trous
`'CO'`	fermeture → ouverture	comble les petits trous avant d'éliminer le bruit externe
`'OCO'`	ouverture → fermeture → ouverture	met l'accent sur l'élimination du bruit externe
`'COC'`	fermeture → ouverture → fermeture	met l'accent sur le comblement des trous et des lacunes

Le paramètre `n` contrôle le nombre d'échelles utilisées par le filtre. À chaque itération i, l'élément structurant est agrandi par somme de Minkowski (`mm.sesum(b, i)`), produisant une séquence de filtres morphologiques de plus en plus englobants. Contrairement à une ouverture ou une fermeture unique avec un grand élément structurant, l'ASF réalise un lissage progressif à plusieurs échelles, préservant mieux la géométrie des objets pertinents tout en éliminant les structures plus petites. Figure 4.6 présente un exemple d'application de l'ouverture, de la fermeture et de l'ASF sur l'image des pièces de monnaie.

```
img_bin    = mm.threshold(img_clahe)
B_disk     = mm.sedisk(19)

img_open  = mm.open(img_bin, B_disk)            # érosion → dilatation
img_close = mm.close(img_bin, B_disk)           # dilatation → érosion
img_oc    = mm.close(img_open, B_disk)          # ouverture suivie de fermeture
img_asf   = mm.asf(img_bin, 'OC', mm.sedisk(3), n=7)
# ASF : disque de base 3×3, croît à chaque itération

mm.show(
    [img_bin, img_open, img_close, img_oc, img_asf],
    titles=["Binarisation Otsu", "Ouverture (A B)", "Fermeture (A B)",
            "Ouverture→Fermeture", "ASF-OC (n=7)"],
    cols=5, figsize=(15, 12)
)
```

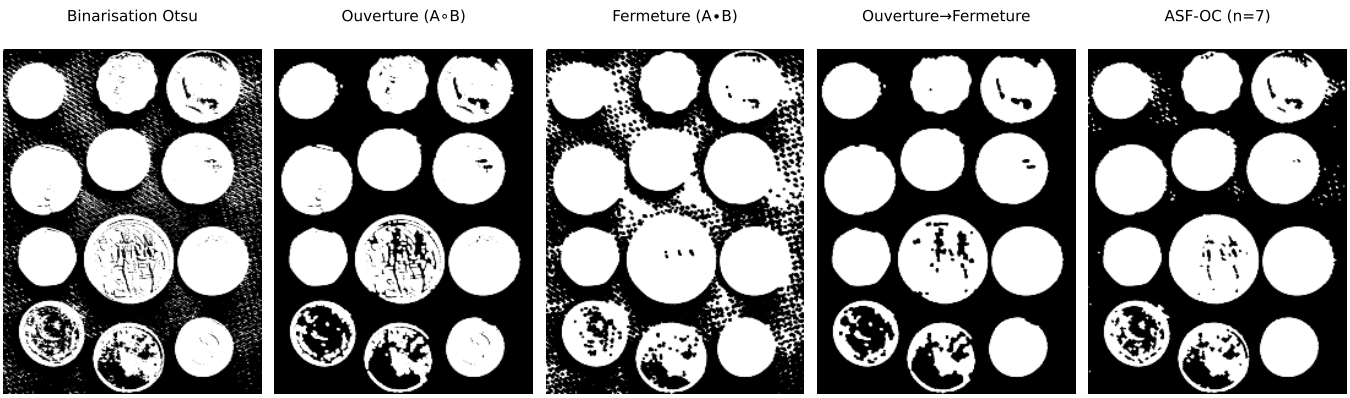


Figure 4.6: Ouverture, fermeture, composition et filtre séquentiel alterné appliqués à la binarisation Otsu des pièces (prétraitées par rehaussement de contraste). Élément structurant : disque 13×13.

4.3.3 Opérateurs Géodésiques

Les opérateurs géodésiques introduisent une contrainte supplémentaire aux opérateurs morphologiques classiques au moyen d'une image de contrôle appelée **masque** g. Au lieu de permettre à l'érosion ou à la dilatation de se propager librement à travers l'image, le résultat de chaque itération est limité point par point par les valeurs du masque, restreignant l'évolution de l'opération aux régions autorisées.

4.3.3.1 Dilatation Géodésique

La **dilatation géodésique** d'une image marqueur f sous une image masque g, à l'aide d'un élément structurant plat b, est définie par :

$$f \oplus_g b = (f \oplus b) \wedge g, \tag{4.7}$$

où $\wedge$ représente le minimum point par point.

En d'autres termes, on effectue d'abord une dilatation conventionnelle sur le marqueur, puis le résultat est restreint par le masque g. Ainsi, la propagation ne peut jamais dépasser les régions autorisées par le masque.

La formulation classique de la dilatation géodésique suppose que le marqueur est contenu dans le masque, c'est-à-dire $f \leq g$, garantissant que l'évolution de l'opération reste toujours limitée par le masque.

4.3.3.2 Implémentation de la dilatation géodésique

La fonction `mm.cdil` implémente directement cet opérateur et permet d'exécuter plusieurs itérations consécutives :

```
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
```

L'instruction `np.minimum(mm.dil(y, b), g)` implémente exactement la définition mathématique de la dilatation géodésique, c'est-à-dire $(y \oplus b) \wedge g$.

Lorsque $n = 1$, la fonction exécute une unique dilatation géodésique. Pour $n > 1$, le résultat de chaque étape devient le marqueur de l'étape suivante, produisant une propagation progressive contrôlée par le masque.

Le simulateur interactif Figure 4.7 permet de suivre, étape par étape, la propagation du marqueur f le long des couloirs du labyrinthe. À chaque itération de `mm.cdil`, le front de dilatation avance vers les cellules voisines libres — celles où $g = 1$ —, tandis que les murs ($g = 0$) restent infranchissables. Le nombre de pas nécessaires pour que le marqueur atteigne la sortie correspond exactement à la longueur géodésique du chemin le plus court à l'intérieur du masque, mettant en évidence la connexion directe entre la dilatation géodésique itérée et la notion de distance dans les graphes.

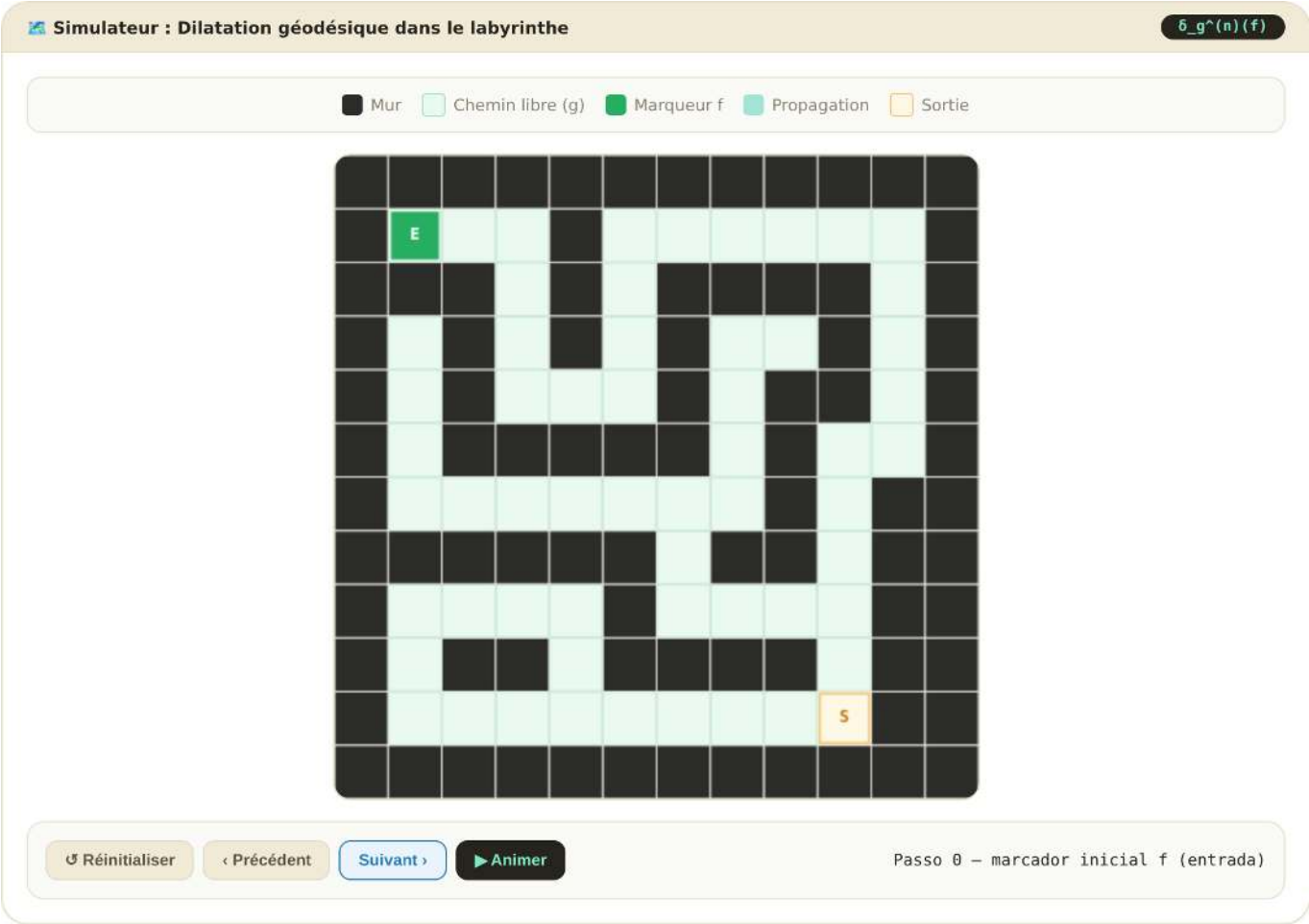
4.3.3.3 Érosion géodésique

De manière duale, l'**érosion géodésique** d'une image marqueur f sous une image masque g est définie par :

$$f \ominus_g b = (f \ominus b) \vee g, \tag{4.8}$$

où $\vee$ représente l'opérateur de maximum point par point.

Dans ce cas, l'érosion conventionnelle du marqueur est suivie d'une contrainte inférieure imposée par le masque. Ainsi, aucun pixel du résultat ne peut prendre une valeur inférieure à celle du pixel correspondant du masque.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-04-cdil.html

Figure 4.7: Simulateur interactif de dilatation géodésique : le marqueur f (vert) se propage pas à pas le long des chemins libres du masque g, sans traverser les murs.

La formulation classique de l'érosion géodésique suppose la condition duale

$$f \geq g,$$

de sorte que le masque agisse comme une limite inférieure tout au long du processus.

4.3.3.4 Implémentation de l'érosion géodésique

La fonction statique `mm.cero` matérialise cet opérateur :

```
staticmethod
def cero(f, g, b=np.zeros((3,3),dtype='uint8'), n=1):
    """Érosion géodésique du marqueur f sous le masque g."""
    y = f.copy()
    for _ in range(n):
        y = np.maximum(mm.ero(y, b), g)
    return y
```

L'instruction `np.maximum(mm.ero(y, b), g)` implémente directement l'expression $(y \ominus b) \vee g$.

Tout comme pour la dilatation géodésique, le paramètre n définit combien d'érosions géodésiques successives seront calculées avant de renvoyer l'image finale.

> **Relation avec la reconstruction morphologique**
>
> La reconstruction morphologique présentée dans la section suivante est obtenue par l'application itérative de la dilatation géodésique (`mm.cdil`) jusqu'à ce qu'un point fixe soit atteint, c'est-à-dire jusqu'à ce qu'aucune cellule ne change de valeur entre deux itérations consécutives. En d'autres termes, la reconstruction consiste en une séquence de dilatations géodésiques successives qui se propagent à l'intérieur du masque jusqu'à ce qu'il n'y ait plus de modifications.
> De manière duale, il est également possible de définir des reconstructions basées sur l'érosion géodésique au moyen d'applications successives de `mm.cero`.

4.3.3.5 Exemple : propagation dans un labyrinthe via dualité

La Figure 4.8 illustre la résolution du problème de connectivité d'un labyrinthe en utilisant la dualité morphologique au moyen des fonctions `mm.cero` et `mm.suprec`.

Au lieu de propager un marqueur à travers les couloirs libres par des dilatations géodésiques, le problème est formulé dans le domaine complémentaire. Initialement, le masque original est inversé,

$$g = 1 - g_{orig},$$

ce qui fait que les murs prennent la valeur 1 et les couloirs la valeur 0. De manière analogue, le marqueur est construit dans ce même domaine complémentaire, contenant une unique valeur 0 à la position de l'entrée du labyrinthe et la valeur 1 sur les autres pixels.

En utilisant l'élément structurant en croix (`mm.secross()`), l'érosion géodésique agit sur le marqueur complémenté. À chaque itération de `mm.cero`, la région connectée au marqueur initial subit des érosions successives, tandis que le masque impose une borne inférieure qui empêche la propagation à travers les murs du labyrinthe.

Les images intermédiaires montrent des états de l'évolution après différents nombres d'itérations (`n=5`, `n=12`, et `n=22`). Le résultat final est obtenu par la reconstruction géodésique par érosion (`mm.suprec`), qui applique des érosions géodésiques successives jusqu'à atteindre un point fixe, c'est-à-dire une situation où aucune modification supplémentaire ne se produit entre deux itérations consécutives.

Dans le domaine complémentaire, la région reconstruite correspond exactement à l'ensemble des couloirs connectés à l'entrée du labyrinthe. Ainsi, la connectivité entre l'entrée et la sortie peut être déterminée directement à partir de l'image reconstruite.

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

# 1. Masque original g (0 = Mur, 1 = Couloir)
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)

# Inversion globale au début (Dualité Morphologique)
g = 1 - g_orig  # Maintenant : 1 = Mur, 0 = Couloir
f = np.ones((10, 10), dtype=np.uint8)
f[0, 1] = 0     # Graine injectée comme 0 dans le couloir

# Élément structurant en croix
B_cruz = mm.secross()

# Traitement direct via mm.cero et mm.suprec dans le domaine inversé
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)

# --- CONFIGURATION DE L'AFFICHAGE GRAPHIQUE ---

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')

# Carte de couleurs adaptée au domaine complémentaire :
# Dans le domaine inversé : 1 = Mur (Bleu Foncé)
# Où g == 0 et image == 1 = Couloir Libre (Gris Clair)
# Où g == 0 et image == 0 = Onde Géodésique Active (Or)
cmap_pipeline = mcolors.ListedColormap(['#ffc13b', '#e0e0e0', '#1e3d59'])

for i, ax in enumerate(axes):
    if i == 0 or i == 1:
        # Pour les conditions initiales (~g et ~f)
        cmap_init = mcolors.ListedColormap(['#e0e0e0', '#1e3d59'])
        ax.imshow(images[i], cmap=cmap_init, vmin=0, vmax=1)
    else:
        # Rendu basé sur les états complémentaires
        render_step = np.zeros_like(g, dtype=np.uint8)
        render_step[g == 1] = 2           # Mur (1 original de la carte de couleurs)
```

```
        render_step[g == 0] = 1            # Couloir par défaut
        render_step[images[i] == 0] = 0    # Onde active (0 original de la carte de couleurs)
        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)

    # Indicateurs graphiques d'Entrée et de Sortie
    ax.plot(1, 0, marker='v', color='#2ecc71', markersize=7, markeredgewidth=1.5)
    ax.plot(8, 9, marker='o', color='#e74c3c', markersize=6, fillstyle='none',
markeredgewidth=2)

plt.tight_layout()
plt.show()
```

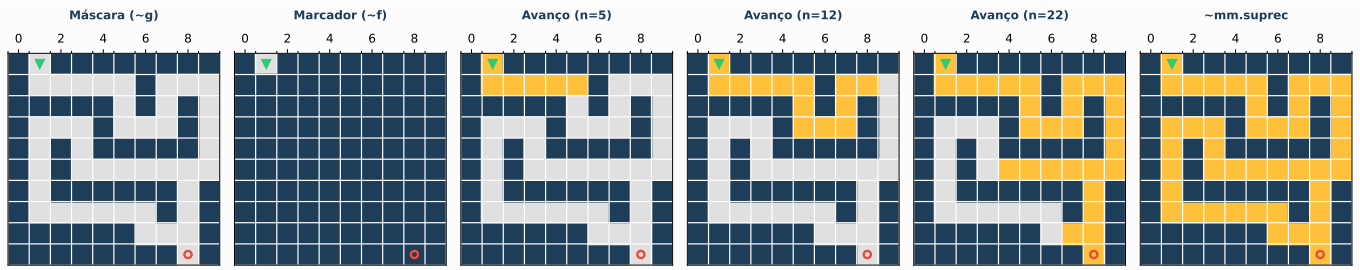


Figure 4.8: Résolution de labyrinthe via Opérations Complémentaires : en inversant le masque et le marqueur au début du *pipeline*, le flux est résolu directement dans le domaine complémentaire via mm.cero et mm.suprec, éliminant les ré-inversions redondantes.

4.3.4 Reconstruction morphologique

La **reconstruction morphologique** propage une image **marqueur** f à l'intérieur d'une image **masque** g, en garantissant que le résultat ne dépasse jamais les valeurs d'intensité imposées par le masque. L'opérateur fondamental qui rend possible cette propagation contenue est la **dilatation géodésique**, définie par la Équation 4.7.

La reconstruction est obtenue par l'application itérative de cette dilatation conditionnée. Initialement, le marqueur est limité par le masque pour établir l'état initial :

$$X^{(0)} = f \wedge g,$$

et les itérations suivantes sont définies de manière récursive par :

$$X^{(k)} = (X^{(k-1)} \oplus b) \wedge g.$$

La séquence croît de manière monotone jusqu'à atteindre un point fixe, produisant la **reconstruction morphologique par dilatation** (également connue dans la littérature sous le nom d'*inf-reconstruction*) :

$$R_g^{\delta}(f) = \lim_{k\to\infty} X^{(k)} = X^{(k)} \quad \text{quand} \quad X^{(k)} = X^{(k-1)}. \tag{4.9}$$

La montée itérative est interrompue dès que la stabilité est atteinte, c'est-à-dire lorsque deux itérations consécutives produisent des matrices avec des valeurs absolument identiques.

4.3.4.1 Implémentation de la reconstruction morphologique

La routine didactique `mm.infrec` implémente directement l'algorithme itératif de point fixe. Initialement, le marqueur effectif initial $X^{(0)}$ est déterminé par l'opération `np.minimum(f, g)`. Pour garantir que la boucle de vérification exécute la première itération sans déclencher de fausses convergences prématurées, la variable de contrôle de l'itération précédente (`y1`) est initialisée avec une valeur sentinelle hors du domaine des données (ou simplement avec une matrice qui force la première exécution).

```
def infrec(f, g, b=np.zeros((3,3), dtype='uint8')):
    """Inf-reconstruction : dilate le marqueur (f   g) jusqu'à convergence sous le masque
    g."""
    y = np.minimum(f, g)
    # Initialise y1 avec des valeurs impossibles pour forcer l'entrée dans la boucle
    y1 = np.full_like(f, 256, dtype=np.int16)
    while not np.array_equal(y, y1):
        y1 = y.copy()
        # Applique la dilatation géodésique : (y   b)   g
        y = np.minimum(mm.dil(y, b), g)
    return y.astype('uint8')
```

À l'intérieur de la boucle `while`, la variable `y` stocke l'estimation courante de la reconstruction $X^{(k)}$, tandis que `y1` préserve l'image de l'étape immédiatement antérieure $X^{(k-1)}$. L'instruction de contrôle conditionnel `np.minimum(mm.dil(y, b), g)` traduit fidèlement la dilatation géodésique théorique, où l'expansion morphologique conventionnelle pilotée par OpenCV est immédiatement « élaguée » et limitée par les barrières d'intensité du masque g. La boucle cesse lorsqu'aucune modification de pixel n'est enregistrée entre les étapes.

4.3.4.2 Avantages de la reconstruction morphologique

La reconstruction morphologique est significativement plus robuste que l'ouverture conventionnelle, car elle est capable de supprimer les structures indésirables sans déformer ou modifier la morphologie des objets qui doivent être préservés.

Alors que l'ouverture classique adoucit les angles, élimine les pointes et déforme les contours en raison de l'imposition géométrique rigide de l'élément structurant, la reconstruction géodésique utilise le masque pour retrouver avec exactitude les limites et les formes originales des objets qui présentent une connectivité avec le marqueur initial.

En termes intuitifs, le marqueur agit comme une graine de contagion qui se propage progressivement, mais uniquement en traversant les régions autorisées par le masque. Les composantes qui n'ont aucune intersection avec le marqueur ne seront jamais reconstruites (étant ainsi éliminées), tandis que les composantes touchées par la graine s'étendent jusqu'à restaurer intégralement leur géométrie d'origine.

Ce comportement discriminatoire et conservateur est illustré à la Figure 4.9, en utilisant l'élément structurant en croix présenté à la Figure 4.3..

```
g = np.array([
    [0,0,0,0,0,0,0,0,0,0],
    [0,1,1,1,0,0,0,1,1,0],
    [0,1,1,1,1,0,0,1,1,0],
    [0,1,1,1,1,0,0,0,0,0],
    [0,1,1,1,1,0,0,0,0,0],
    [0,1,1,1,0,0,0,0,0,0],
    [0,0,1,1,1,0,0,1,0,0],
    [0,0,1,1,1,0,0,1,1,0],
    [0,0,0,1,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0]], dtype=np.uint8) * 255

B_cruz = mm.secross()
f = mm.ero(g, mm.sebox())          # marqueur : noyau de l'objet 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"  Stabilité à l'itération 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)
)
```

```
Stabilité à l'itération 5 : True
```

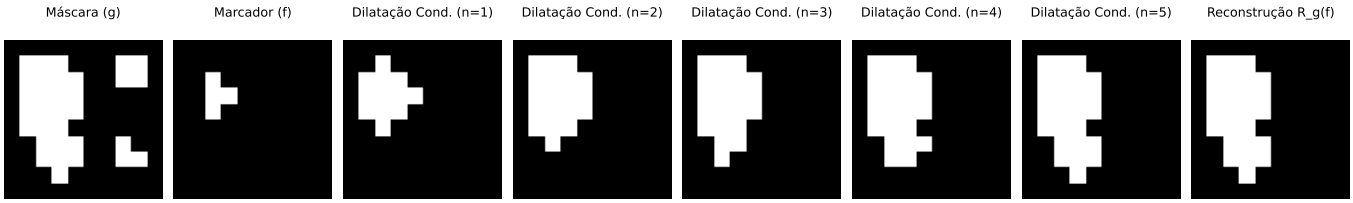


Figure 4.9: *Pipeline* de Reconstruction morphologique par Dilatation conditionnée : le masque contient deux objets, le marqueur isole uniquement le noyau de l'objet principal, et les itérations reconstruisent sa forme exacte jusqu'à la convergence.

4.3.5 Remplissage des trous et suppression des bords

Deux opérateurs basés sur la reconstruction morphologique complètent le *pipeline* de nettoyage binaire. Leurs principales caractéristiques sont résumées dans la Table 4.4.

Remplissage des trous (`mm.clohole`) supprime les cavités entièrement entourées par l'objet, quelle que soit leur taille, sans modifier les contours externes. La procédure agit sur le complément de l'image, en utilisant comme marqueur une restriction du cadre (*frame*) au fond :

$$\text{clohole}(f) = \left(R^{\delta}_{f^c}(\text{frame}(f) \wedge f^c)\right)^c \tag{4.10}$$

Sur le plan opérationnel, on reconstruit d'abord le fond externe, puis on applique la complémentation pour récupérer les objets dont les trous ont été remplis.

Suppression des objets de bord (`mm.edgeoff`) élimine tous les objets qui touchent le bord de l'image, en ne préservant que les composantes entièrement internes. Le marqueur est obtenu par l'intersection entre le cadre (*frame*) et les objets de l'image :

$$\text{edgeoff}(f) = f \setminus R^{\delta}_{f}(\text{frame}(f) \wedge f) \tag{4.11}$$

Tableau 4.4: Comparaison entre les opérateurs *clohole* et *edgeoff*.

Opérateur	Marqueur	Masque	Effet
`mm.clohole`	*frame* restreint au fond (f^c)	f^c	Remplit les trous internes
`mm.edgeoff`	*frame* restreint à l'objet (f)	f	Supprime les objets connectés au bord

L'évolution pas à pas de ces transformations géodésiques peut être suivie dans les figures suivantes. La Figure 4.10 illustre le mécanisme d'inondation contrôlée de l'opérateur `mm.clohole`, dans lequel la reconstruction se fait à partir du fond externe et empêche la propagation vers les régions internes non connectées à l'extérieur, ce qui aboutit au remplissage cohérent des cavités internes. En revanche, la Figure 4.11 détaille la dynamique de l'opérateur `mm.edgeoff`, où seules les composantes connectées au bord sont reconstruites puis supprimées, ne préservant que les objets entièrement contenus à l'intérieur de l'image.

La connectivité de la propagation géodésique est contrôlée par l'élément structurant : `mm.sebox()` (voisinage de 8) inclut les connexions diagonales, tandis que `mm.secross()` (voisinage de 4) les exclut. Par conséquent, le choix de l'élément structurant affecte les composantes atteintes par la reconstruction et, donc, celles qui seront préservées ou supprimées.

4.3.5.1 Conformité avec l'implémentation

Les définitions ci-dessus sont directement alignées avec l'implémentation dans `morph.py`, reproduite ci-dessous :

```
staticmethod
def clohole(f, b=np.ones((3,3),dtype='uint8')):
    # marqueur restreint au fond de l'image
    marqueur = mm.frame(f, border=1) & mm.neg(f)
    return mm.neg(mm.infrec(marqueur, mm.neg(f), b))

staticmethod
def edgeoff(f, b=np.ones((3,3),dtype='uint8')):
    # marqueur restreint aux objets de l'image
    marqueur = mm.frame(f, border=1) & f
    return mm.subm(f, mm.infrec(marqueur, f, b))
```

Ces implémentations rendent explicite le fait que les deux opérateurs sont des instances directes de la reconstruction morphologique par dilatation géodésique avec `mm.infrec`, ne différant que par le choix du marqueur et du masque : `clohole` agit sur le complément de l'image, tandis que `edgeoff` agit directement sur le domaine des objets.

```
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)       # borda externa
```

```
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"  Validação 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
)
```

```
Validação clohole : True
```

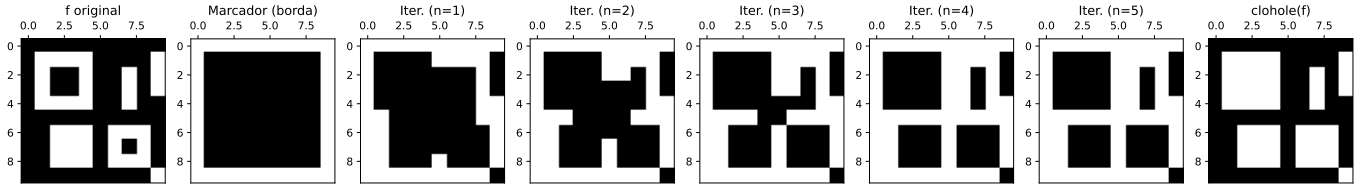


Figure 4.10: *Pipeline* de preenchimento de buracos (*clohole*) : o marcador vem da borda da imagem, restrito ao complemento f^c. A dilatação geodésica reconstrói o fundo externo ; após a complementação, os buracos internos ficam preenchidos.

```
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)
)
```

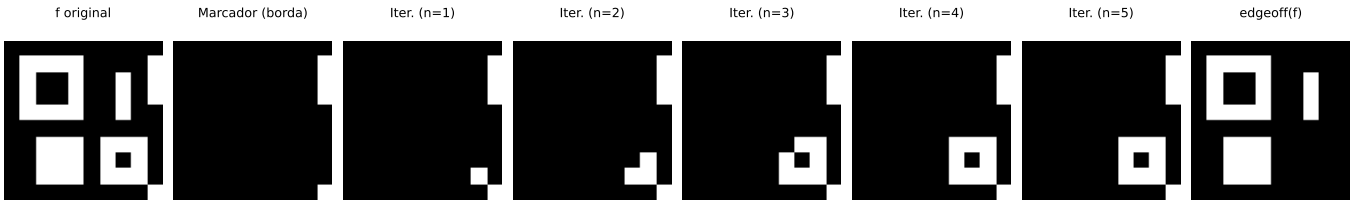


Figure 4.11: *Pipeline* de suppression des structures de bordure (*edgeoff*) : le marqueur capture les racines connectées aux extrémités, la reconstruction délimite ces éléments et la soustraction ne préserve que les objets totalement internes.

4.3.6 Pipeline de Limpeza Binária com CLAHE

Com base na análise anterior — na qual o CLAHE produziu o maior valor da variância interclasses ($\sigma_B^2 \approx 2{,}47 \times 10^3$), ver Figure 4.2, e o operador `mm.clohole` mostrou-se eficaz no preenchimento das cavidades internas —, o *pipeline* final de segmentação, ilustrado na Figure 4.12, é estruturado pelo seguinte fluxo computacional:

$$\text{gray} \xrightarrow{\text{CLAHE}} \xrightarrow{\text{Otsu}} \xrightarrow{\text{open}} \xrightarrow{\text{clohole}} \xrightarrow{\text{open}} \xrightarrow{\text{edgeoff}} \text{segmentação}$$

Após a etapa de `mm.clohole`, aplica-se uma segunda abertura morfológica com um elemento estruturante maior (`mm.sedisk(33)`, disco de diâmetro 33). Essa operação remove pequenas regiões residuais e artefatos que possam ter permanecido após a segmentação. Em particular, o preenchimento geodésico pode transformar pequenas cavidades isoladas em componentes conectados ao objeto, tornando conveniente uma etapa adicional de filtragem baseada em tamanho. O diâmetro foi escolhido de modo que as moedas permaneçam capazes de conter o elemento estruturante, enquanto componentes significativamente menores sejam eliminados.

Nesta imagem, nenhuma moeda está conectada à borda da matriz. Consequentemente, a aplicação de `mm.edgeoff` não altera o resultado obtido após a segunda abertura. Ainda assim, essa etapa é mantida no *pipeline* por robustez, pois em outras imagens podem existir objetos parcialmente visíveis ou conectados às bordas, que devem ser removidos antes da etapa de análise.

Pourquoi l'ouverture après le clohole ?

L'opérateur `mm.clohole` remplit toutes les cavités fermées présentes dans les objets segmentés. Dans certaines situations, de petites régions indésirables peuvent subsister après cette étape ou devenir connectées aux objets principaux. L'ouverture morphologique subséquente supprime les composants plus petits que l'élément structurant, tout en préservant les pièces en raison de leur taille significativement plus grande.

```
#   Prétraitement : uniquement CLAHE
img_clahe0 = mm.clahe(img_coins_gray, 2.0, 8)

#   Étape 1 : Binarisation Otsu
img_bin = mm.threshold(img_clahe0)

#   Étape 2 : Ouverture - supprime les bruits blancs de fond
img_open = mm.open(img_bin, mm.sedisk(9))

#   Étape 3 : clohole - ferme tous les trous internes
img_hole = mm.clohole(img_open)

#   Étape 4 : Ouverture (grand noyau) - supprime les artefacts du clohole
img_limpo = mm.open(img_hole, mm.sedisk(33))

#   Étape 5 : edgeoff - supprime les objets touchant le bord
```

```
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",                "Ouverture (r=9)",
            "clohole",          "Ouverture (r=33)",     "Final (edgeoff)"],
    cols=6, figsize=(18, 6)
)
```

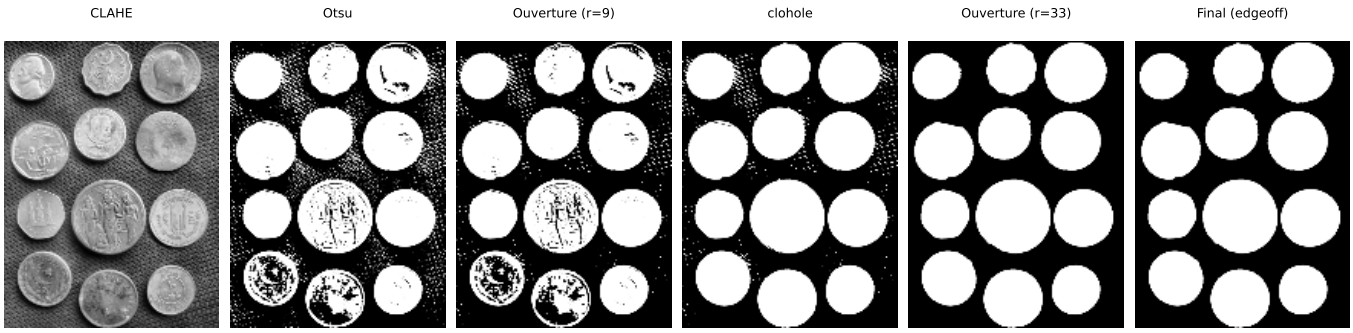


Figure 4.12: *Pipeline* complet de segmentation avec CLAHE : Otsu → ouverture (r=9) → clohole → ouverture (r=33) → edgeoff.

4.3.7 Morphologie en niveaux de gris

Les opérateurs morphologiques s'étendent naturellement aux images en niveaux de gris. Dans cette formulation, l'érosion et la dilatation agissent directement sur les niveaux d'intensité de l'image. Pour des éléments structurants plats ($b \equiv 0$), l'érosion correspond au **minimum local** et la dilatation au **maximum local** dans le voisinage défini par l'élément structurant.

L'interprétation intuitive est simple : l'érosion **assombrit** les régions en remplaçant chaque pixel par la plus petite valeur présente dans son voisinage, tandis que la dilatation **éclaircit** les régions en utilisant la plus grande valeur disponible. La combinaison de ces opérateurs permet de construire des transformations capables de rehausser les contours, de supprimer les tendances d'éclairage et de mettre en évidence les structures locales.

Trois opérateurs dérivés sont particulièrement utiles :

Gradient morphologique — met en évidence les contours comme la différence entre la dilatation et l'érosion :

$$\text{grad}_B(f) = (f \oplus B) - (f \ominus B) \tag{4.12}$$

Top-hat — met en évidence les structures brillantes plus petites que l'élément structurant (différence entre l'image originale et son ouverture) :

$$\text{top-hat}_B(f) = f - (f \circ B) \tag{4.13}$$

Black-hat — met en évidence les structures sombres plus petites que l'élément structurant (différence entre la fermeture et l'image originale) :

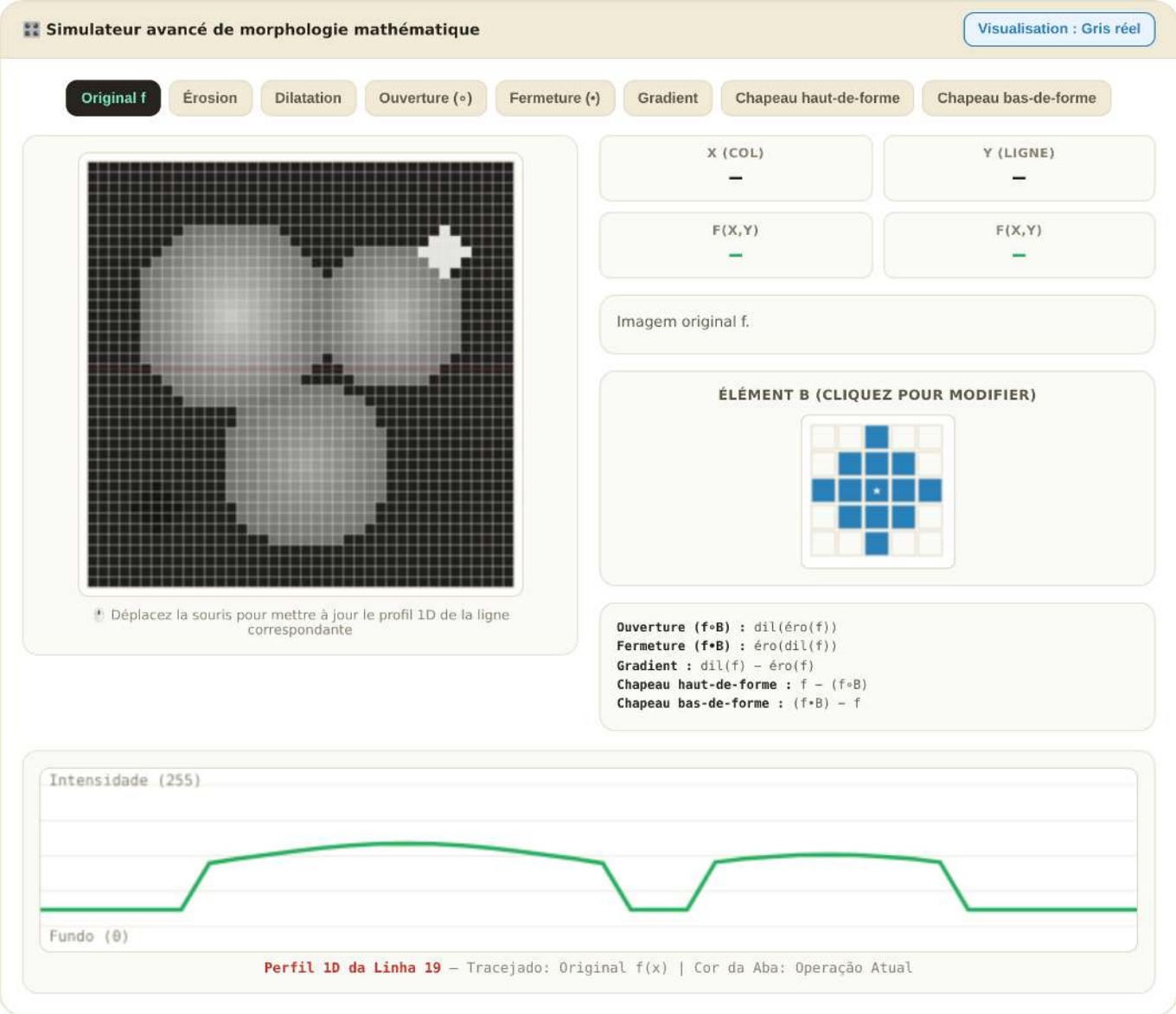
$$\text{black-hat}_B(f) = (f \bullet B) - f \tag{4.14}$$

Le *Top-hat* extrait les détails brillants qui ne survivent pas à l'ouverture, tandis que le *Black-hat* révèle les détails sombres supprimés par la fermeture. Quant au gradient morphologique, il rehausse les transitions abruptes d'intensité, produisant une représentation similaire à celle d'un détecteur de contours.

Pour comprendre le mécanisme de ces opérateurs au niveau local, la Figure 4.13 présente un simulateur interactif de morphologie en tons de gris. Le simulateur permet de modifier librement l'élément structurant, de visualiser son déplacement sur l'image et de suivre simultanément le profil unidimensionnel des intensités. Ainsi, il devient possible d'observer directement comment l'érosion sélectionne les minima locaux, comment la dilatation sélectionne les maxima locaux et comment le gradient morphologique émerge de la différence entre ces deux opérateurs.

Le bouton situé dans le coin supérieur droit permet d'alterner entre la visualisation originale en tons de gris et une représentation en fausses couleurs (*colormap*) pour les trois types de gradients uniquement. La version colorée facilite la perception visuelle des variations d'intensité, rendant plus évidente l'action des opérateurs morphologiques sur les maxima, les minima et les transitions locales de l'image.

Les opérateurs présentés sont disponibles dans `morph.py` via les fonctions `mm.gradm`, `mm.tophat` et `mm.blackhat`.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-04-operadores-av.html

Figure 4.13: Simulateur interactif avancé de morphologie avec élément structurant modifiable et profil 1D.

La Figure 4.14 illustre les effets de ces opérateurs sur l'image des pièces et leurs histogrammes respectifs.

On observe que l'érosion déplace la distribution vers des intensités plus faibles, tandis que la dilatation la déplace vers des intensités plus élevées. Le gradient concentre les valeurs dans les régions de contour, et les opérateurs *Top-hat* et *Black-hat* produisent des histogrammes fortement concentrés à de faibles niveaux d'intensité, car seules de petites structures locales sont mises en évidence.

```
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)
    # CORRECTION : range utilise len(h) dynamiquement pour correspondre au retour de la
    bibliothèque
    ax.bar(range(len(h)), h, color='steelblue', width=1, edgecolor='steelblue')
    ax.set(xlim=(0, 255)); plt.tight_layout()
    return fig2img(fig)

# 1. Traitement morphologique de 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. Assemblage dynamique de la paire : [Image, Histogramme]
imgs, titles = [], []
for nome, img in operadores:
    imgs += [img, plot_hist(img, f"Hist. {nome}")]
    titles += [nome, f"Hist. {nome}"]

# 3. Affichage final en grille de deux colonnes (Image | Histogramme)
mm.show(imgs, titles=titles, cols=4, figsize=(10, 8))
```

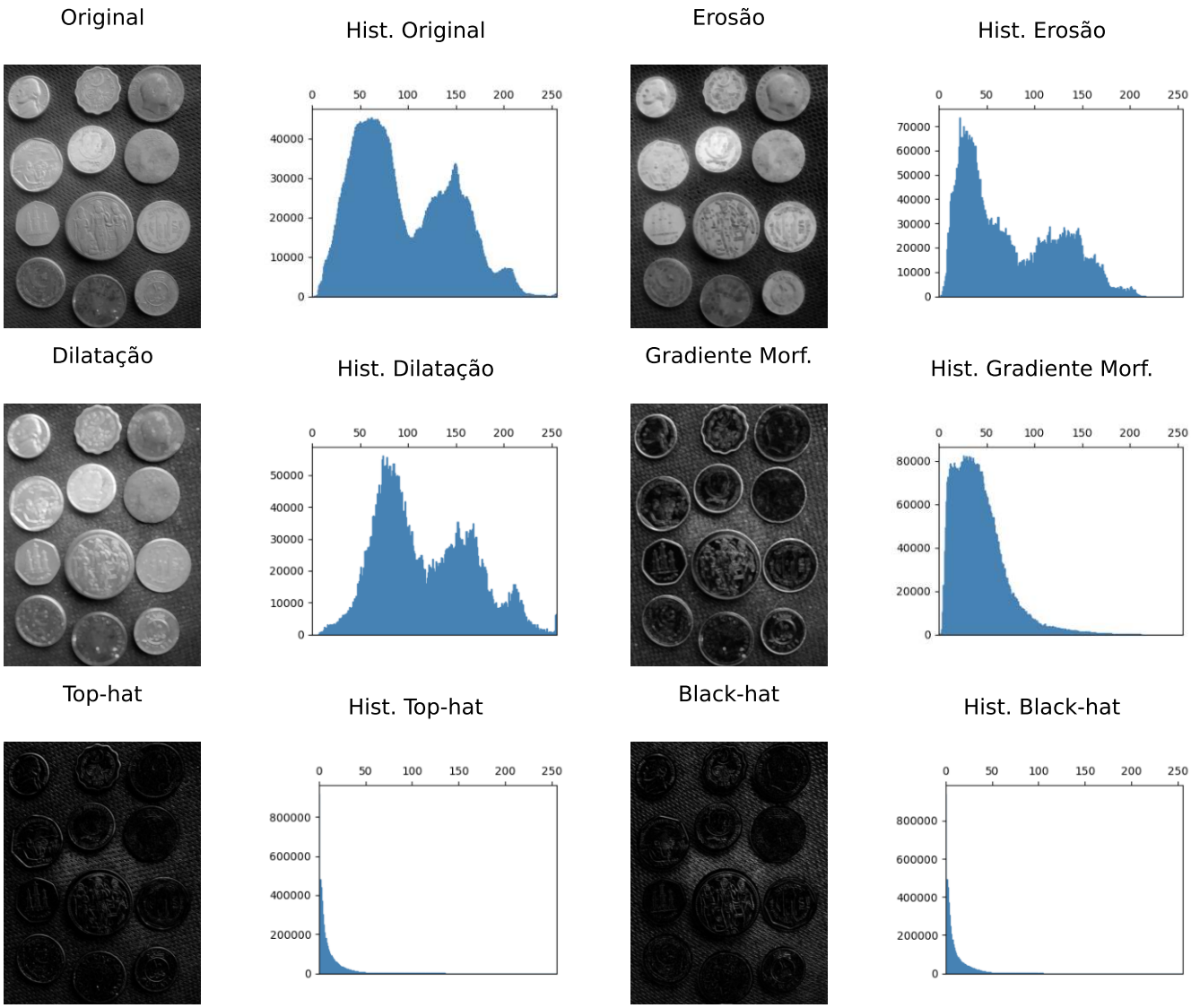


Figure 4.14: Morphologie en niveaux de gris et leurs histogrammes respectifs. L'érosion réduit les intensités locales, la dilatation les amplifie, le gradient met en évidence les bords, et les opérateurs Top-hat et Black-hat révèlent les détails locaux clairs et sombres. Élément structurant : disque de diamètre 19.

Les opérateurs morphologiques présentés précédemment seront désormais utilisés comme outils de raffinement et de génération de marqueurs pour des méthodes de segmentation plus avancées, présentées ci-après.

4.4 Segmentation d'Images : Fondements et Taxinomie

La **segmentation d'images** consiste à diviser l'image en régions associées à des objets ou à des structures d'intérêt. En TNI, elle représente la transition entre le traitement de bas niveau — comme le filtrage et le rehaussement — et des étapes d'analyse plus avancées, telles que l'extraction de caractéristiques, la reconnaissance et l'interprétation de la scène.

Formellement, l'objectif de la segmentation consiste à décomposer le domaine spatial complet d'une image, noté Ω, en une partition de sous-ensembles $\{R_1, R_2, \dots, R_n\}$ qui satisfait simultanément les critères de **complétude** et de **disjonction** :

$$\bigcup_{i=1}^{n} R_i = \Omega, \qquad R_i \cap R_j = \emptyset \quad \forall\, i \neq j \tag{4.15}$$

Outre les propriétés de complétude et de disjonction exprimées dans la Équation 4.15, chaque sous-région R_i doit constituer un domaine **homogène** selon un prédicat de similarité défini sur des propriétés locales — intensité, couleur ou texture — et, simultanément, être **distincte** des régions adjacentes.

Les techniques de segmentation peuvent être organisées en différentes familles. Dans ce chapitre, l'accent sera mis sur les approches résumées dans la Table 4.5, fondées principalement sur des critères d'intensité, de connectivité et de proximité spatiale.

Tableau 4.5: Taxinomie simplifiée des principales approches de segmentation et d'affinement étudiées dans ce chapitre.

Approche	Critère de Segmentation	Opérateurs de Référence
Seuillage	Partitionnement de l'espace des intensités	Critère d'Otsu, seuillage global et local
Morphologie Mathématique	Relations spatiales définies par des éléments/fonctions structurants	Érosion, dilatation, ouverture, fermeture et reconstruction
Basée sur les Régions	Homogénéité locale et connectivité spatiale	Étiquetage des composantes connexes, Transformée de Distance et *Watershed*

Jusqu'à ce point, le développement pratique s'est concentré sur le **seuillage**, au moyen de la combinaison entre l'égalisation adaptative CLAHE et la méthode globale d'Otsu. Cette étape a été complétée par des opérateurs de **reconstruction morphologique** basés sur des dilatations géodésiques, implémentés par les fonctions `mm.infrec`, `mm.clohole` et `mm.edgeoff`, produisant un masque binaire propre et adapté à l'analyse.

Cependant, dans les scénarios où des objets distincts apparaissent connectés dans le masque binaire — que ce soit par contact physique, chevauchement partiel ou par de minces ponts de pixels produits par la segmentation —, le seuillage cesse d'être suffisant pour individualiser chaque objet. Dans ces cas, de multiples objets composent une seule composante connexe, ce qui complique les étapes ultérieures de mesure et d'interprétation.

Pour surmonter cette limitation, les prochaines sections introduisent trois outils complémentaires : l'**Étiquetage des Composantes Connexes**, la **Transformée de Distance** et l'algorithme de segmentation par ***Watershed*** basé sur des marqueurs. Ensemble, ces techniques permettent de séparer les objets adjacents, d'identifier les régions individuellement et d'extraire des descripteurs géométriques cohérents pour une analyse quantitative.

4.4.1 Étiquetage

Le **étiquetage de composantes connexes** (*connected component labeling*) est l'opérateur qui attribue un identifiant entier unique à chaque ensemble de pixels appartenant à une même composante connexe dans une image binaire.

> **Définition formelle**
>
> Étant donné une image binaire f et une relation de connectivité définie par un élément structurant B (typiquement connectivité-4 ou connectivité-8), l'algorithme d'étiquetage de la Figure 4.15 produit une image g dans laquelle tous les pixels appartenant à la même composante connexe reçoivent le même étiquette entier positif, tandis que les pixels appartenant à des composantes distinctes reçoivent des étiquettes différentes.

La connectivité définit quels pixels sont considérés comme voisins directs d'un pixel (x, y). Les définitions les plus couramment utilisées sont :

- **Connectivité-4 :** ne considère que les quatre voisins orthogonaux (nord, sud, est et ouest).

- **Connectivité-8 :** considère les quatre voisins orthogonaux ainsi que les quatre voisins diagonaux, totalisant huit voisins.

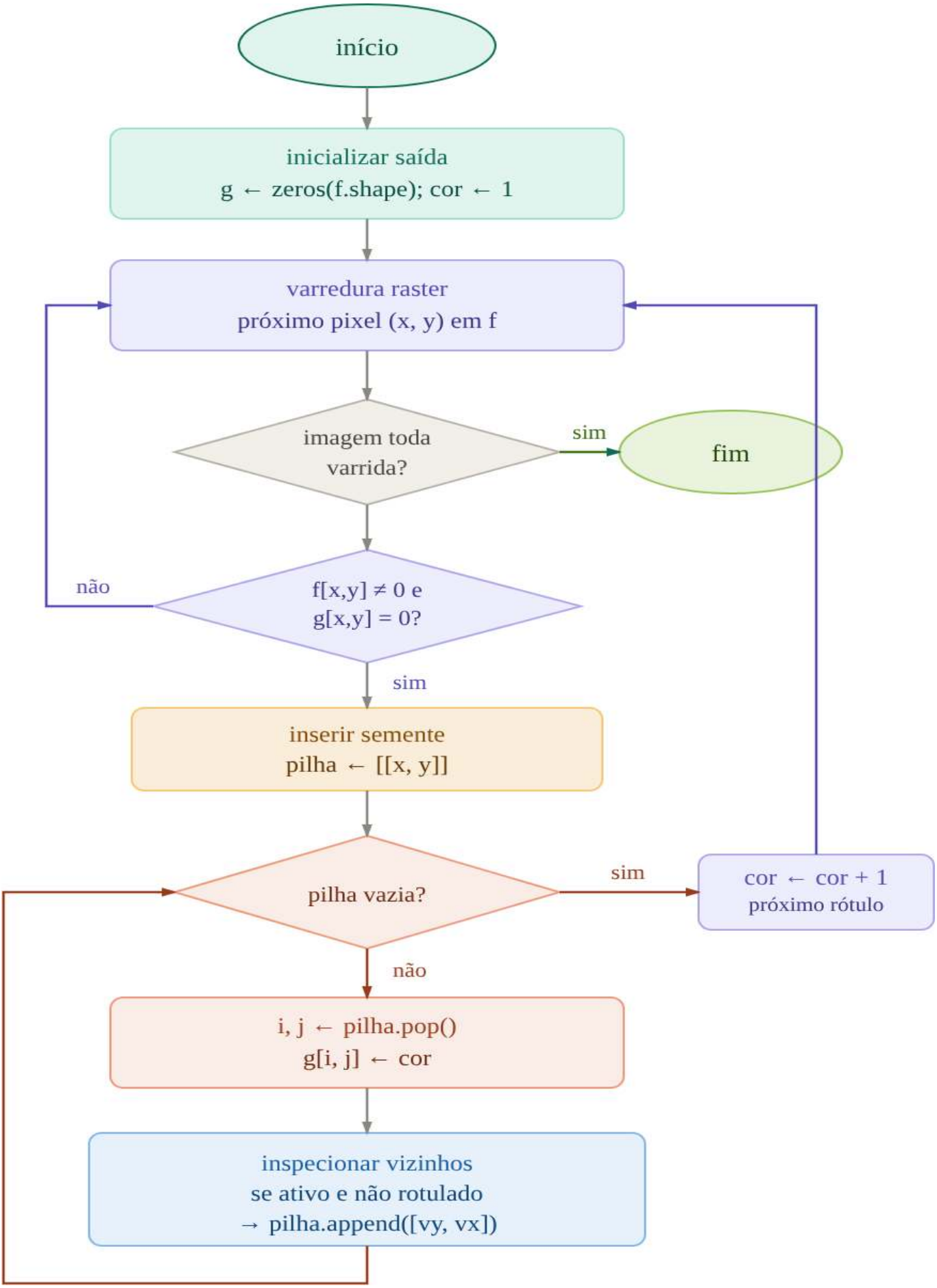
Le choix de la connectivité influence directement la formation des composantes connexes et, par conséquent, le résultat de l'étiquetage, comme illustré dans la Figure 4.17. Un exemple supplémentaire peut être exploré de manière interactive dans le simulateur présenté dans la Figure 4.16.

L'implémentation dans `morph.py` propose deux versions de cet opérateur. La fonction `mm.label0` reproduit explicitement l'algorithme de *flood-fill* à l'aide d'une pile et permet de contrôler la connectivité via l'élément structurant adopté. Quant à `mm.label`, elle délègue l'opération à l'implémentation optimisée d'OpenCV (`mm.label0`). Dans les deux cas, le résultat est une image étiquetée dans laquelle chaque composante connexe reçoit un identifiant entier distinct.

L'auxiliaire `_viz` itère sur la fenêtre structurante `b` centrée en (i, j), ne générant que les voisins valides à l'intérieur des limites de l'image — la connectivité souhaitée est entièrement déterminée par la forme de `b` passée à l'algorithme.

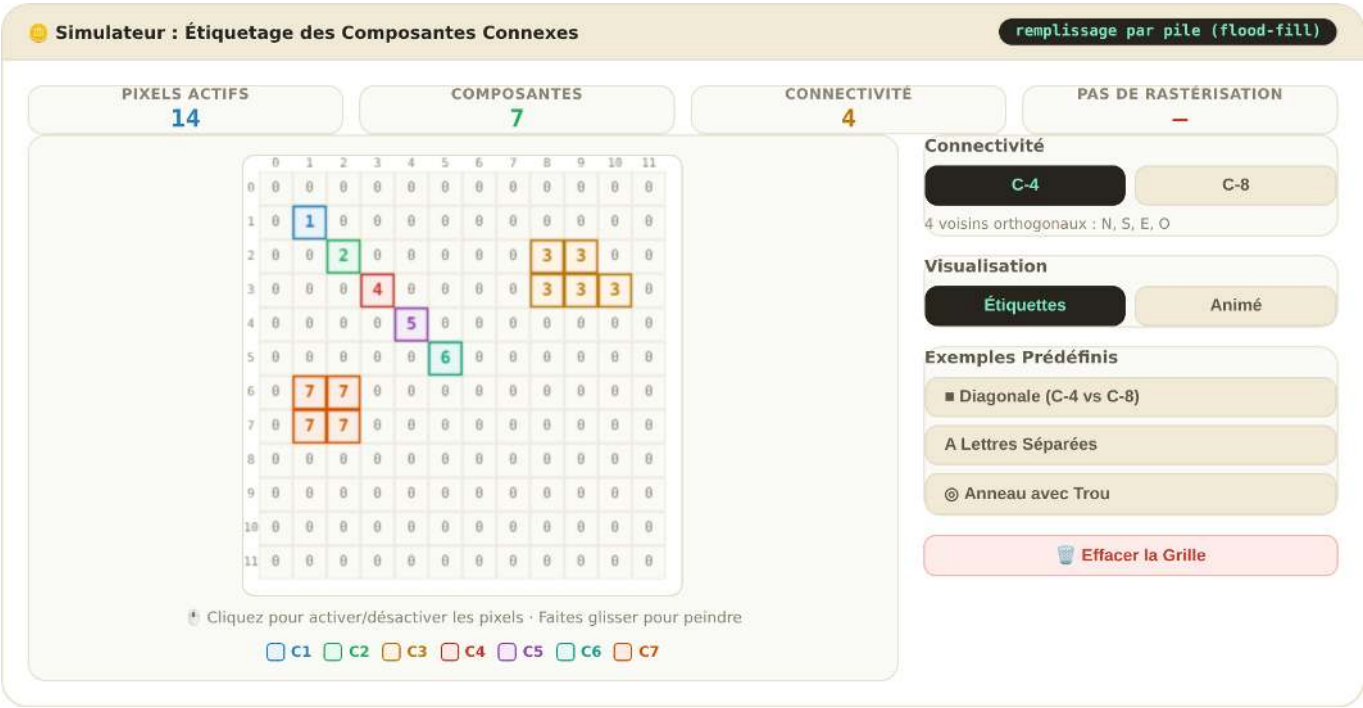
Exemple didactique — effet de la connectivité :

```
# Image binaire 10×10 avec des composantes adjacentes en diagonale
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],  # diagonale avec la ligne précédente
    [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],  # diagonale avec la ligne précédente
    [0,0,0,0,0,0,0,0,0,0]], dtype=np.uint8) * 255

# Élément structurant en croix (connectivité-4) et carré (connectivité-8)
B4 = mm.secross()   # connectivité-4
B8 = mm.sebox()     # connectivité-8

# Étiquetage avec label0 (didactique) et 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)

# Validation croisée
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"   Composantes C4 : {lbl4_cv2.max()}  |  Composantes C8 : {lbl8_cv2.max()}")

# Normalisation pour visualisation
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 originale",
        f"Étiquetage C4\n({lbl4_cv2.max()} composantes)",
        f"Étiquetage C8\n({lbl8_cv2.max()} composantes)"
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-alg-rotulagem2.html

Figure 4.15: Algorithme d'étiquetage par *flood-fill* avec pile.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-04-rotulacao.html

Figure 4.16: Simulateur interactif d'étiquetage de composantes connexes (*connected component labeling*) : visualisation de l'expansion flood-fill, connectivité-4 et connectivité-8.

```
    ],
    cols=3, figsize=(12, 4), axis=True
)
```

```
label0 (C4) == cv2 (C4) : True
label0 (C8) == cv2 (C8) : True
 Composantes C4 : 7  |  Composantes C8 : 4
```

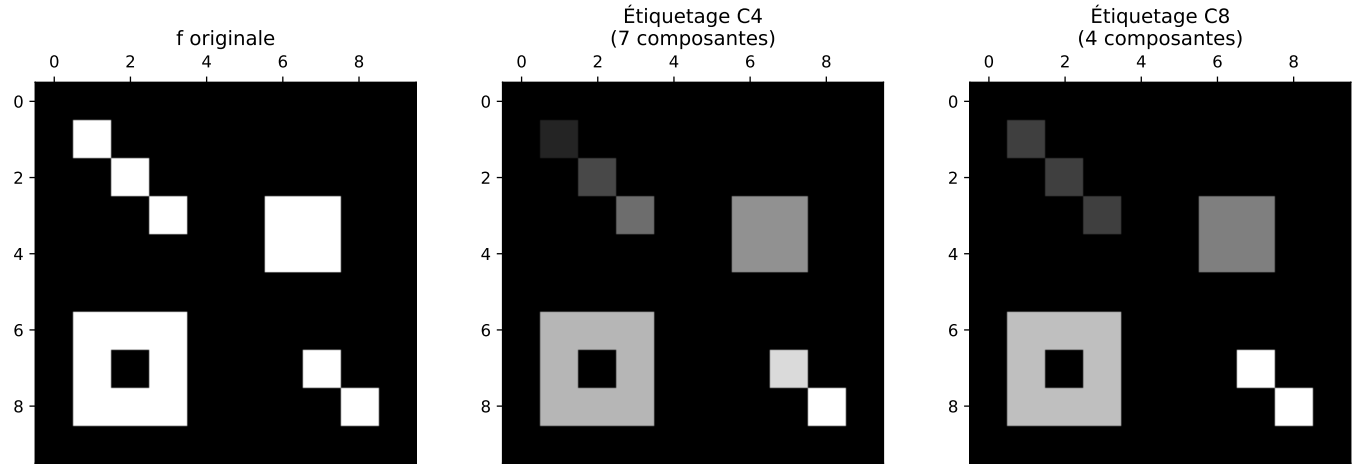


Figure 4.17: Effet de la connectivité sur l'étiquetage : l'image binaire 10×10 contient des pixels adjacents en diagonale. Avec la connectivité-4, ces pixels forment des composantes distinctes ; avec la connectivité-8, ils fusionnent en une seule composante.

4.4.2 Transformée de Distance

La **Transformée de Distance** (TD) est un opérateur qui, appliqué à une image binaire f, produit une image en niveaux de gris D dans laquelle chaque pixel appartenant à l'objet ($f(x, y) \neq 0$) reçoit comme valeur la distance géométrique jusqu'au pixel de fond ($f(x', y') = 0$) le plus proche :

$$D(x, y) = \min_{(x',y') \,:\, f(x',y')=0} d\big((x, y),\, (x', y')\big)$$

où $d(\cdot, \cdot)$ est une métrique de distance — typiquement la distance euclidienne (L_2). Le résultat est une représentation topographique des objets : les pixels situés à l'intérieur prennent des valeurs élevées, tandis que les pixels proches des bords présentent de faibles valeurs de distance. Les **maxima locaux** de D correspondent aux points les plus éloignés du bord de l'objet, souvent proches de leurs centres géométriques ou de leurs centres d'inscription maximale — propriété particulièrement utile pour la génération automatique de marqueurs dans l'algorithme *watershed*.

> **ℹ Définition formelle utilisant des érosions**
>
> La TD admet également une interprétation morphologique itérative, selon l'algorithme de la Figure 4.18. Considérons un élément structurant b dont la valeur centrale est nulle et dont les voisins possèdent des coûts négatifs associés au déplacement. En appliquant des érosions successives avec cet élément structurant particulier, les valeurs des pixels des objets (qui doivent prendre la distance maximale possible de l'image) sont progressivement réduites selon les coûts définis par b. La valeur accumulée de cette propagation représente alors la distance au fond selon la métrique induite par l'élément structurant.

Cette interprétation est implémentée dans `mm.dist1()`, qui accumule des érosions successives en utilisant l'opération `mm.ero1()`. Quant à `mm.dist()`, elle délègue le calcul de la distance euclidienne à l'opérateur optimisé d'OpenCV `mm.dist(f)`, où `f` est l'image binaire d'entrée, la distance L2 spécifie la métrique euclidienne (L_2) et `5` indique l'utilisation d'un masque 5×5 pour approximer la distance avec une grande précision.

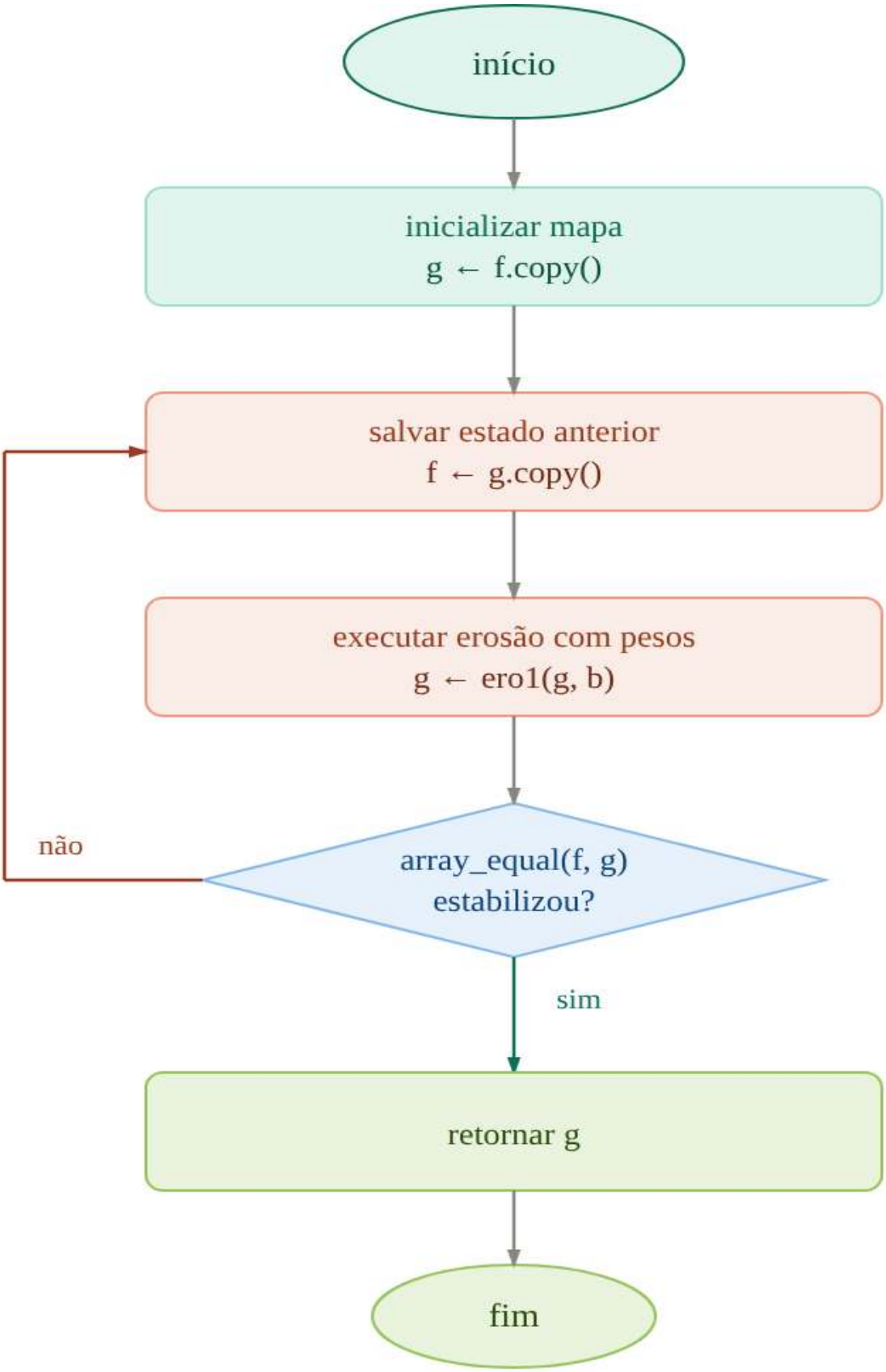
La fonction `dist1` produit une transformée de distance discrète dont la métrique est déterminée par la géométrie et les poids de l'élément structurant utilisé. Par exemple, en utilisant un élément structurant en croix avec un coût unitaire pour les quatre voisins orthogonaux, on obtient la distance de ***Manhattan*** (L_1). D'autres choix de voisinage et de poids induisent des métriques différentes. Quant à `mm.dist()`, elle calcule une approximation efficace de la distance euclidienne (L_2).

En raison de la nécessité d'érosions successives sur toute l'image, l'approche `dist1` présente un coût computationnel significativement plus élevé que `mm.dist()`, étant utilisée dans ce livre principalement à des fins pédagogiques et pour mettre en évidence la relation entre la morphologie mathématique et les transformées de distance.

A Figure 4.19 apresenta um simulador interativo da TD: é possível posicionar o cursor sobre diferentes pixels do objeto e observar, em tempo real, o valor da distância associado àquela posição, isto é, a distância até o pixel de fundo mais próximo. A Figure 4.20 apresenta um exemplo prático dessa execução em ambiente Python.

A Figure 4.19 apresenta um simulador interativo da transformada de distância (TD): é possível posicionar o cursor sobre diferentes pixels do objeto e observar, em tempo real, o valor da distância associado a essa posição, ou seja, a distância até o pixel de fundo mais próximo. A Figure 4.20 apresenta um exemplo prático dessa execução em ambiente Python.

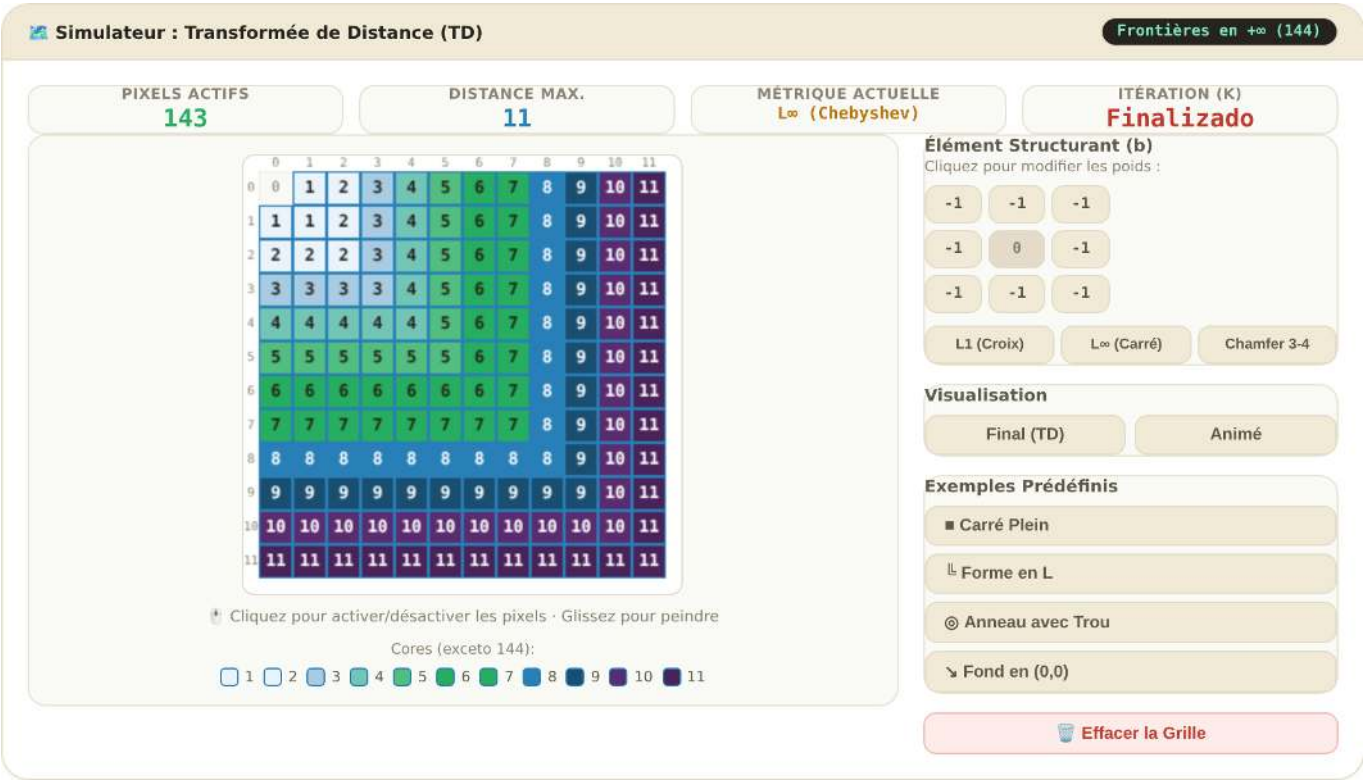
```
f = np.ones((10, 10), dtype=np.uint8) * 255
f[0, 0] = 0

B_cruz = np.array([
    [-np.inf, -1, -np.inf],
```



Ponto Fixo Numérico: A distância emerge da propagação matemática do valor zero.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-alg-distancia2.html

Figure 4.18: Algorithme de la Transformée de Distance.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-04-distancia.html

Figure 4.19: Simulateur interactif de la Transformée de Distance (TD) itérative via érosion en niveaux de gris. Les pixels hors de l'image prennent la valeur maximale (144), propageant les coûts depuis le fond interne.

```
    [-1,        0, -1],
    [-np.inf, -1, -np.inf]
], dtype=float)

d_iter = mm.dist1(f, B_cruz)
d_l2   = mm.dist(f)

print(f"Max. dist1 (érosions) : {int(d_iter.max())} px")
print(f"Max. dist  (L2)       : {int(d_l2.max())} px")

mm.show(
    [f, d_iter, d_l2],
    titles=["f original", "mm.dist1 (érosions avec croix)", "mm.dist (L2)"],
    cols=3, figsize=(12, 4), axis=True
)
```

```
Max. dist1 (érosions) : 18 px
Max. dist  (L2)       : 12 px
```

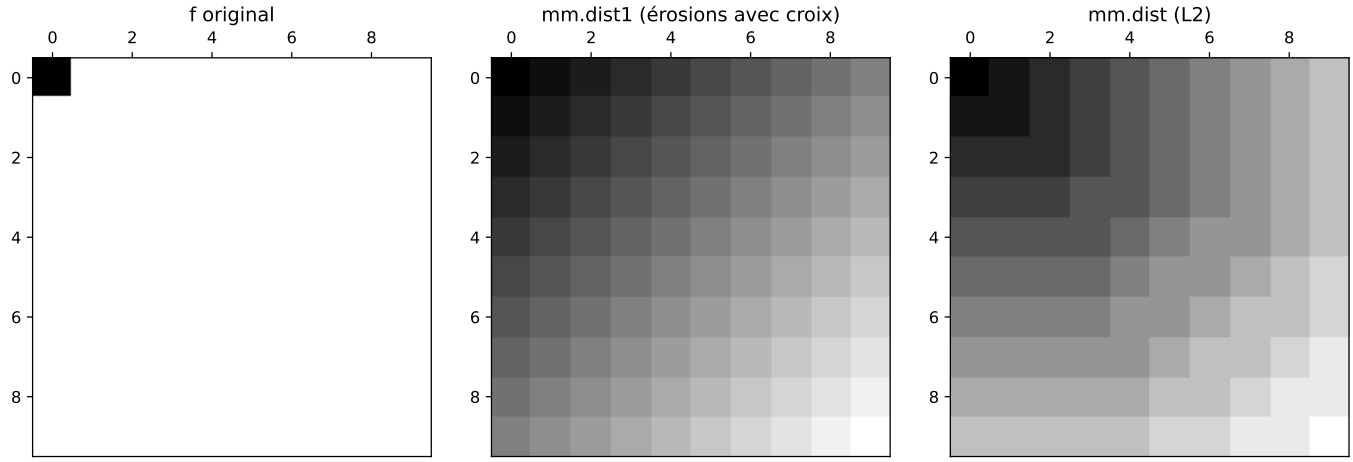


Figure 4.20: Transformée de distance sur image binaire 10×10. À gauche : originale (*foreground* = 255). Au centre : mm.dist1 itérative (érosions avec croix). À droite : mm.dist (L2).

L'annotation des valeurs numériques directement sur les pixels permet de vérifier comment `dist1` propage les distances selon la métrique induite par la fonction structurante utilisée. Dans le cas de l'élément en croix à coût unitaire, les valeurs obtenues correspondent à la distance de *Manhattan* (L_1). Bien que `dist1` et `mm.dist` produisent des valeurs numériques distinctes en adoptant des métriques différentes, les deux transformées préservent la structure topographique des objets, faisant en sorte que leurs maxima se produisent dans des régions centrales similaires. Cette propriété justifie l'utilisation de `mm.dist` dans des applications pratiques, en raison de sa grande efficacité computationnelle.

4.4.3 Transformada de Distância Euclidiana em quatro passos

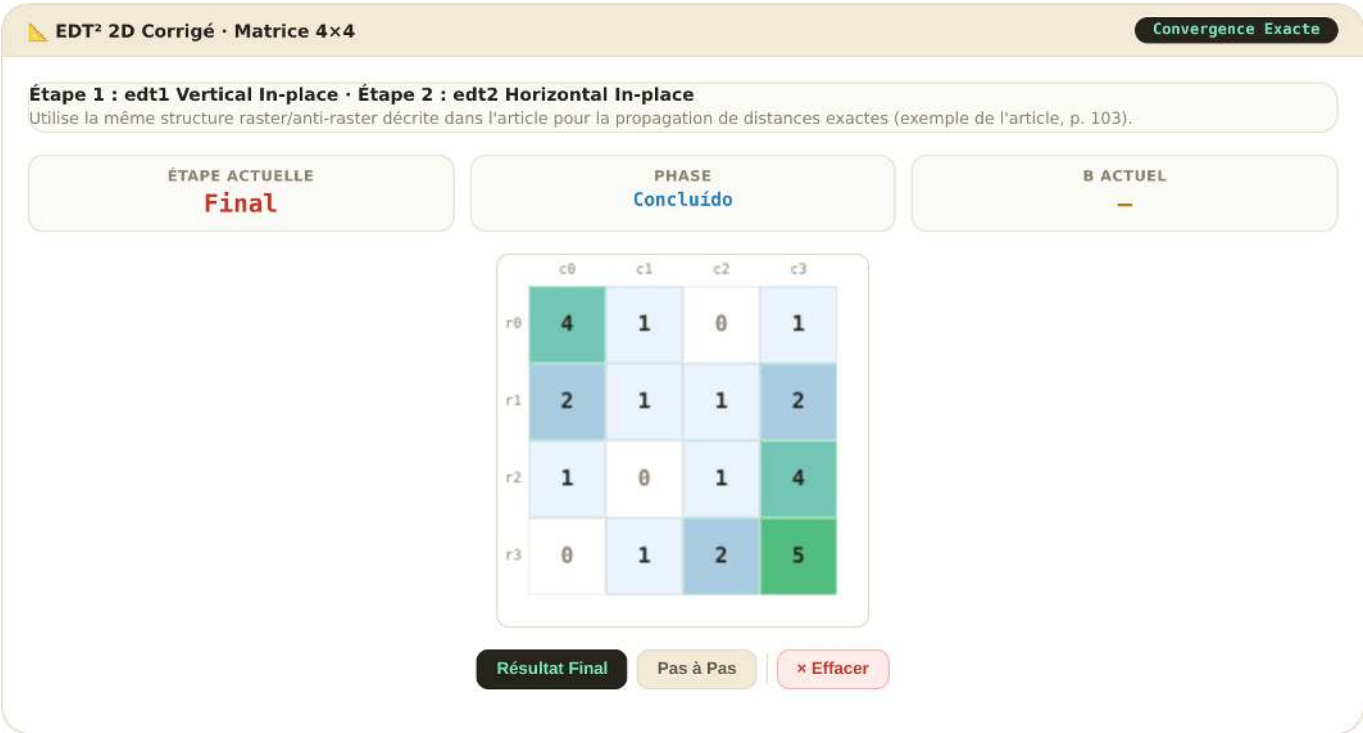
Le simulateur de la Figure 4.21 implémente l'algorithme de la TDE de Lotufo (2001) en deux étapes. Dans la première étape, la fonction `edt1` effectue une transformation unidimensionnelle verticale de manière séquentielle (*in-place*), parcourant chaque colonne en *raster* (↓) et en *anti-raster* (↑) pour calculer les distances dans la direction verticale (deux étapes : Sud et Nord). Dans la deuxième étape, la fonction `edt2` utilise ce résultat comme entrée et effectue une propagation horizontale par files : pour chaque ligne de la matrice, deux files de priorité, `Eq` et `Wq`, sont initialisées en parcourant les indices de colonne dans des directions opposées (`Eq` de `W-1` à `1`, `Wq` de `2` à `W`), de sorte que chaque pixel mis à jour met immédiatement en file ses voisins pour retraitement dans la même itération (deux étapes supplémentaires : Est et Ouest). Ce mécanisme de file permet d'appliquer des érosions successives avec des poids impairs croissants (`b = 1, 3, 5, ...`, incrémenté à chaque itération de la boucle externe) sans que la propagation ne reste piégée dans des valeurs obsolètes, car chaque itération résout complètement la chaîne de dépendances horizontale avant l'incrément suivant de `b`. La convergence de cette propagation produit la Transformada de Distância Euclidiana sur toute la matrice, combinant l'information verticale obtenue dans `edt1` avec la propagation horizontale par file réalisée dans `edt2`.

4.4.3.1 Transformée de Distance Géodésique

La transformée de distance géodésique associe à chaque pixel la distance minimale jusqu'à un marqueur, sous la contrainte imposée par un masque. Ainsi, la propagation s'effectue exclusivement à travers les pixels autorisés, préservant la connectivité du domaine.

La Figure 4.22 illustre ce processus dans un labyrinthe : (a) le masque `g` ; (b) la distance géodésique `D1` calculée depuis l'entrée ; (c) la distance `D2` calculée depuis la sortie ; et (d) le chemin minimal obtenu à partir de ces deux transformées.

Le chemin optimal est déterminé par la somme des distances (`D1 + D2`). Les pixels appartenant à la trajectoire minimale sont ceux pour lesquels cette somme atteint sa plus petite valeur, définissant une connexion entre l'entrée et la sortie avec une longueur géodésique minimale.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-04-tde.html

Figure 4.21: Simulateur interactif 2D (4x4) avec synchronisation stricte des files de propagation horizontale (b) pour obtenir la convergence exacte décrite dans l'article.

Ce principe permet de résoudre des labyrinthes sans avoir à explorer explicitement toutes les possibilités de parcours. La solution émerge directement de la propagation des distances dans un domaine restreint. Cette approche est particulièrement pertinente pour les labyrinthes de haute complexité, comme ceux construits à partir de structures quasicristallines et de cycles hamiltoniens décrits par Singh (2024).. Dans Zampirolli (2025), ce même formalisme est employé pour résoudre un labyrinthe complexe ; ensuite, la méthode est illustrée sur une version simplifiée du problème.

```
import numpy as np

# 1 = couloir, 0 = mur
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)

# marqueur de l'entrée
entrada = np.zeros_like(g, dtype=np.uint8)
entrada[0,1] = 1

# marqueur de la sortie
saida = np.zeros_like(g, dtype=np.uint8)
saida[9,8] = 1
```

```
# distances géodésiques
D1 = mm.gdist(g, entrada)
D2 = mm.gdist(g, saida)

# somme des distances
S = D1 + D2

# plus petite valeur valide de la somme
dmin = np.min(S[S > 0])

# pixels appartenant à un chemin optimal
caminho = (S == dmin)

print("Distance géodésique minimale :", dmin)

mm.show(
    [g, D1, D2, caminho],
    titles=[
        "Labyrinthe",
        "Distance de l'Entrée",
        "Distance de la Sortie",
        f"Plus court chemin\n(d={dmin})"
    ],
    cols=4,
    figsize=(14,4),
    axis=True
)
```

```
Distance géodésique minimale : 15
```

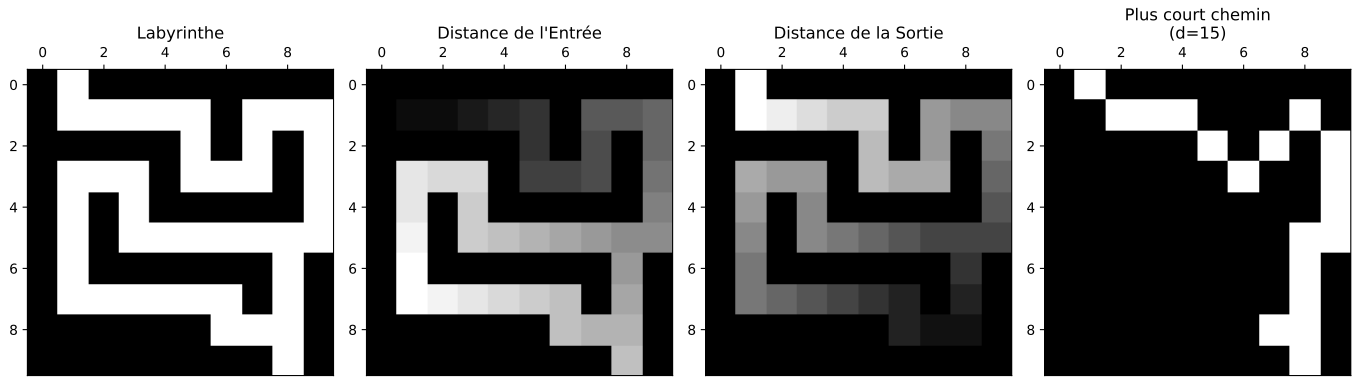


Figure 4.22: Plus court chemin géodésique dans un labyrinthe. Les distances géodésiques sont calculées à partir de l'entrée et de la sortie en utilisant mm.gdist. Les pixels dont la somme des deux distances est égale à la distance minimale entre les marqueurs appartiennent à un chemin optimal.

4.4.4 Segmentation par *Watershed*

L'algorithme ***Watershed*** interprète une image en niveaux de gris comme une surface topographique, où les valeurs élevées correspondent à des montagnes et les valeurs faibles à des vallées ou bassins de drainage (*catchment basins*). Dans le contexte de la segmentation basée sur des marqueurs, les maxima de la Transformée de Distance sont souvent utilisés pour identifier des régions internes aux objets, fournissant ainsi des graines fiables pour le processus d'inondation.

La segmentation est ensuite réalisée par une simulation conceptuelle d'inondation progressive à partir de ces marqueurs. Au fur et à mesure que les bassins associés à différentes graines s'étendent, les régions voisines finissent par entrer en contact. À cet instant, des barrières virtuelles, appelées *lignes de partage des eaux* (*watershed lines*), sont construites et délimitent les objets de la scène. Ce mécanisme permet de séparer des

objets adjacents ou partiellement superposés, même lorsqu'ils forment un seul composant connexe après le seuillage.

L'implémentation didactique présentée dans ce chapitre explore d'abord le concept de croissance de régions (*region growing*) confinée par un masque binaire, comme détaillé dans l'algorithme interactif de la Figure 4.23.

> **Version didactique versus implémentation classique**
>
> La fonction `mm.`watershed0' n'implémente pas l'algorithme *watershed* classique. Son objectif est d'illustrer, de manière simplifiée, la propagation des marqueurs par croissance de régions (*region growing*), permettant ainsi de visualiser comment différentes graines se disputent l'occupation de l'espace disponible. La croissance est délimitée par un masque binaire de support et contrôlée par un mécanisme de stagnation, générant un résultat semblable à une partition de Voronoi restreinte à la géométrie des objets d'entrée.
> Quant à la fonction `mm.watershed`, elle utilise l'implémentation optimisée d'OpenCV (`mm.watershed`), qui réalise l'inondation sur une surface topographique définie par l'image d'entrée. Dans ce cas, la propagation des marqueurs est influencée par les valeurs des pixels, ce qui fait que les lignes de séparation se forment naturellement sur les crêtes du relief.

Figure 4.24 présente un simulateur itératif qui illustre la propagation des marqueurs à travers la région d'intérêt. Chaque marqueur agit comme une source d'inondation qui étend sa zone d'influence jusqu'à rencontrer des régions provenant d'autres graines. Dans l'algorithme d'OpenCV, les pixels appartenant aux lignes de partage sont identifiés par la valeur `-1`, représentant les frontières entre bassins adjacents.

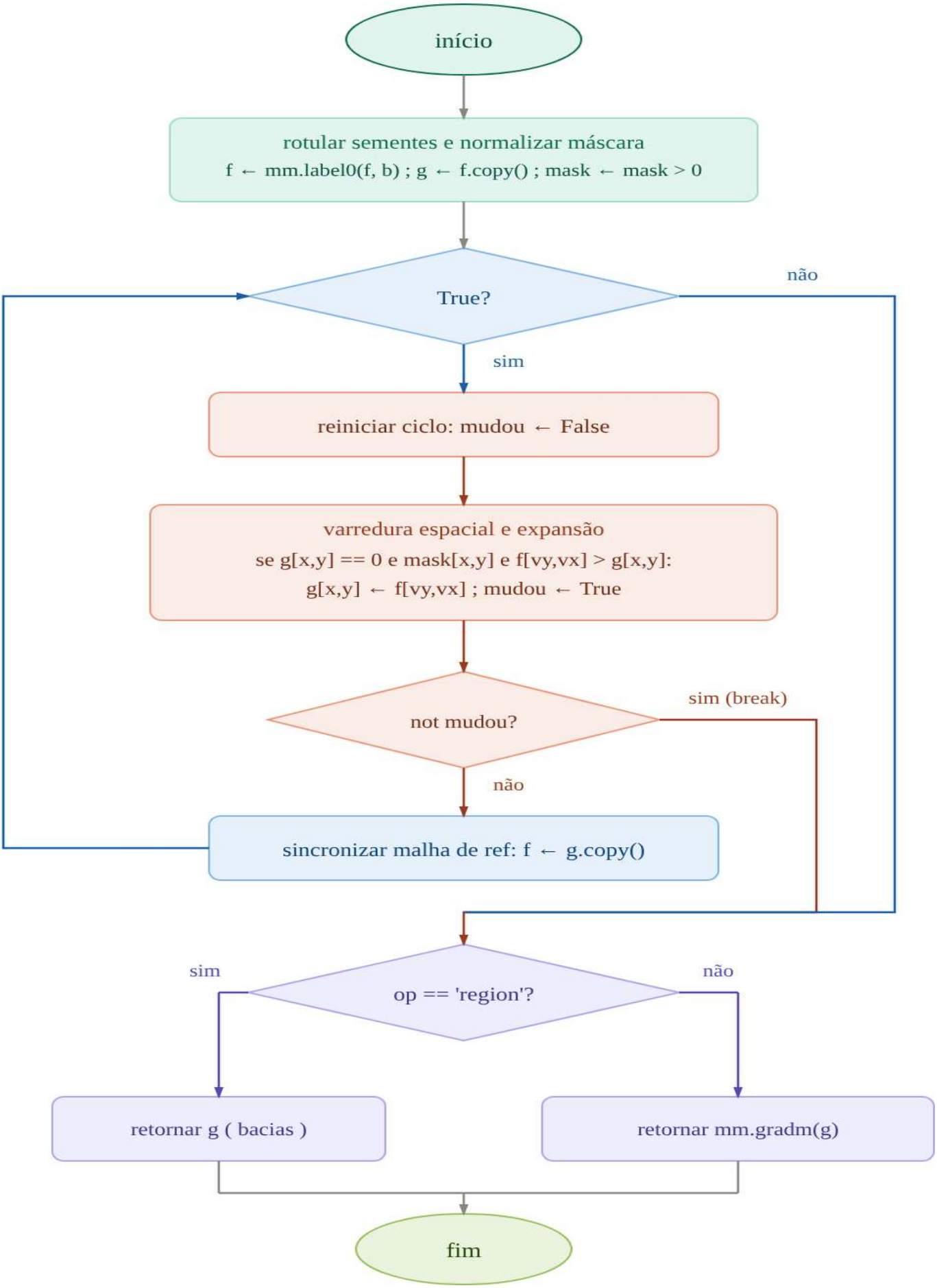
Pipeline morphologique du *watershed*

Le *watershed* basé sur des marqueurs intègre généralement un flux plus large de segmentation. Dans les images réelles, des étapes de prétraitement sont souvent nécessaires pour améliorer le contraste, réduire le bruit et générer des marqueurs fiables. Ce flux complet est résumé dans la Table 4.6.

Tableau 4.6: *Pipeline* complet du *watershed* basé sur des marqueurs pour les images réelles.

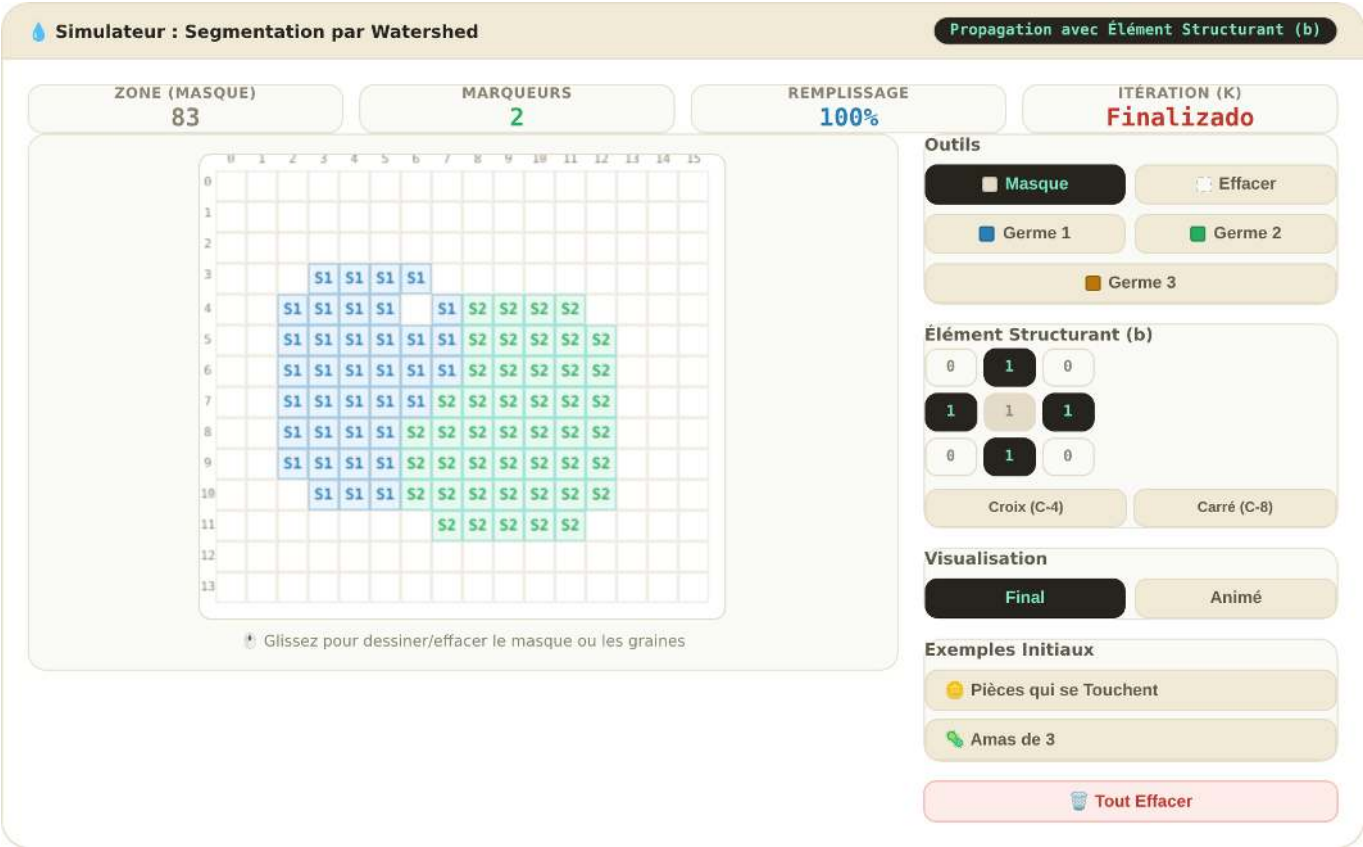
Étape	Opération	Finalité
1	CLAHE + Lissage	Amélioration du contraste et réduction du bruit
2	Seuillage	Séparation initiale entre l'objet et le fond
3	Ouverture/Fermeture	Suppression du bruit et des petites imperfections
4	Dilatation du masque	Identification du fond certain (*Sure Background*)
5	Transformée de distance + Seuil	Identification de l'objet certain (*Sure Foreground*)
6	Région incertaine	Différence entre le fond certain et l'objet certain
7	`mm.watershed`	Propagation des marqueurs à travers la région incertaine

Pour mettre en avant exclusivement les concepts de transformée de distance, de marqueurs et d'inondation topographique, l'exemple de la Figure 4.25 utilise une image binaire synthétique et adopte un flux simplifié, résumé dans la Table 4.7.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-alg-watershed2.html

Figure 4.23: Algorithme didactique de *Watershed* par croissance de régions limitée par masque.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-04-watershed.html

Figure 4.24: Simulateur interactif de l'algorithme *Watershed* par propagation morphologique. Dessinez le masque, positionnez les marqueurs et ajustez l'élément structurant pour observer l'inondation. Lorsque les bassins se rencontrent simultanément, l'égalité est résolue en supposant l'une des régions de manière aléatoire.

Tableau 4.7: *Pipeline* simplifié utilisé dans l'exemple didactique de la Figure 4.25.

Étape	Opération	Finalité
1	Transformée de distance	Construction de la surface topographique
2	Seuil de la TD	Extraction des marqueurs (*Sure Foreground*)
3	Dilatation	Détermination du fond certain (*Sure Background*)
4	Région incertaine	Différence entre le fond et les marqueurs
5	`mm.watershed`	Propagation des marqueurs et génération des frontières

```
# 1. Image synthétique et Transformée de Distance
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. Marqueurs (pics de la distance)
m = (dist > 0.8 * dist.max()).astype(np.uint8) * 255

# 3. Exécution du Watershed0 conditionnel (m=marqueurs d'abord, 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. Affichage des résultats
mm.show([f_sint, dist, m, w_reg, w_line], cols=5, figsize=(16, 4),
        titles=["Original", "Distance", "Marqueurs", "Régions", "Lignes"])
```

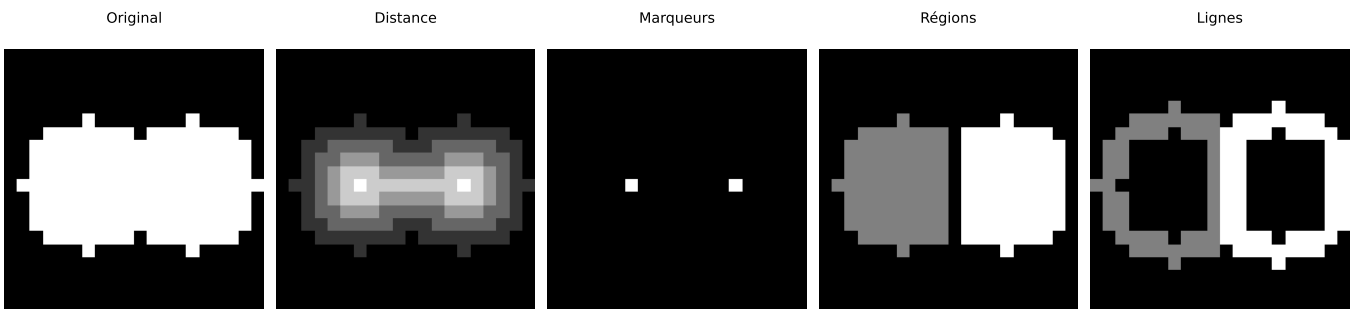


Figure 4.25: *Pipeline watershed* délimité par masque sur image binaire 20×20.

Application aux pièces superposées :

```
img_base = img_coins_gray

# 1. Simuler des pièces superposées/connectées (en utilisant la masque binaire de base)
img_sobrepostas = mm.dil(img_final, mm.sebox(40))
```

```
# 2. Ouverture morphologique pour nettoyer les bruits
opening = mm.open(img_sobrepostas, mm.sebox(2))

# 3. Transformée de distance
dist = mm.dist(opening)
dist_vis=(255*(dist/dist.max())).astype(np.uint8) if dist.max() > 0 else dist.astype(np.uint8)

# 4. Pics sûrs (Marqueurs des pièces)
picos = (dist > 0.5 * dist.max()).astype(np.uint8) * 255

# 5. Exécution du Watershed (Ajusté pour utiliser la nouvelle signature)
# Nous passons 'opening' directement dans mask, car elle délimite la portée d'expansion des
pièces
ws_region = mm.watershed(picos, mask=opening, op='region')
ws_line   = mm.watershed(picos, mask=opening, op='line')

# 6. Comptage des objets (Ignore le fond 0)
labels = np.unique(ws_region)
labels = labels[labels > 0]
print(f"Objets détectés : {len(labels)}")

# 7. Annotation finale
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)

# Dessine les lignes de séparation en rouge
mask_ann = mm.dil(ws_line, np.ones((11, 11), np.uint8))
img_annotated[mask_ann > 0 ] = [255, 0, 0]

# Affichage
mm.show(
    [img_base, img_sobrepostas, dist_vis, picos, ws_region, img_annotated],
    titles=["Original", "Superposées", "Distance", "Marqueurs", "Watershed", "Annoté"],
    cols=3, rows=2, figsize=(12, 8)
)
```

```
Objets détectés : 12
```

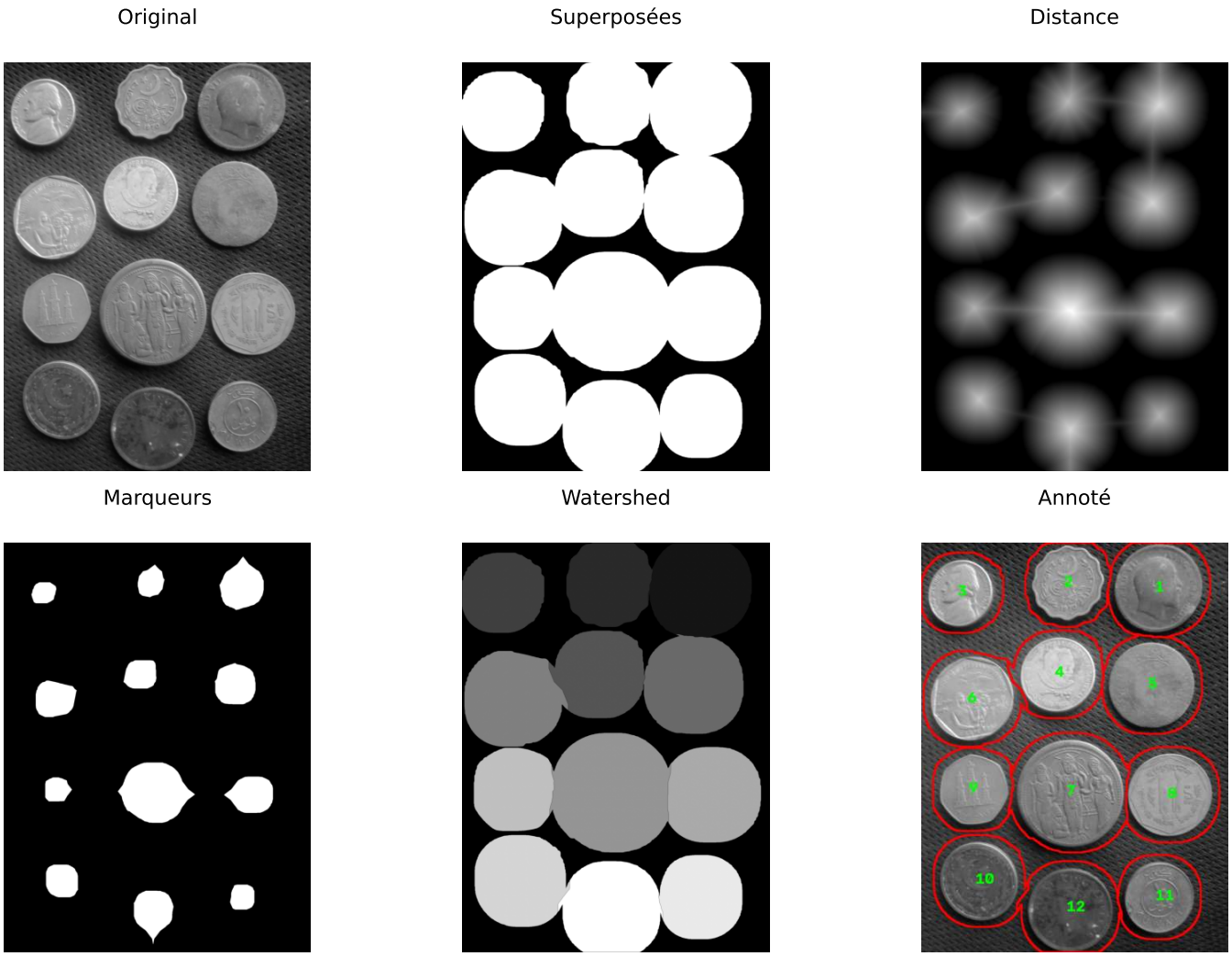


Figure 4.26: *Pipeline Watershed* pour séparation de pièces superposées : de la masque binaire dilatée jusqu'aux contours finaux annotés sur l'image originale.

4.5 Extraction de Composants et Descripteurs de Forme

Après la segmentation et le raffinement morphologique, l'étape suivante consiste à identifier individuellement chaque objet présent dans l'image et à extraire ses propriétés géométriques. Cette étape est fondamentale pour les tâches de mesure, de classification et de reconnaissance de formes.

Un **composant connexe** est un ensemble maximal de pixels appartenant à l'objet qui restent mutuellement connectés selon une relation de connectivité préalablement définie (connectivité 4 ou 8). Après l'étiquetage, chaque composant reçoit un identifiant unique, permettant d'analyser ses caractéristiques individuellement.

OpenCV propose deux approches complémentaires pour cette analyse, résumées dans la Table 4.8.

Tableau 4.8: Comparaison entre les approches basées sur les composants connexes et sur les contours.

	`connectedComponentsWithStats`	`findContours`
Retourne	étiquette par pixel et statistiques par composant	séquence de points décrivant le contour
Descripteurs directs	aire, *boîte englobante* et centroïde	périmètre, forme et hiérarchie
Objets en contact	tend à fusionner les régions connectées	tend à produire un unique contour externe

	`connectedComponentsWithStats`	`findContours`
Usage typique	comptage, filtrage et étiquetage	analyse géométrique et descripteurs de forme

4.5.1 Rotation et statistiques des composants

La fonction `mm.label0` réalise simultanément l'étiquetage des composants connexes et l'extraction de descripteurs de base pour chaque région. L'opérateur retourne :

- une image d'étiquettes (`labels`) ;
- des statistiques géométriques (`stats`) ;
- les coordonnées des centroïdes (`centroids`).

Les statistiques comprennent l'aire, la largeur, la hauteur et la position de la *bounding box* minimale alignée sur les axes de l'image.

La Figure 4.27 illustre l'application de cet opérateur après le *pipeline* de segmentation des pièces de monnaie. Chaque composant connexe reçoit une couleur distincte et son aire est annotée directement sur l'image.

```
# 1. Étiquetage et statistiques
n, labels, stats, centroids = cv2.connectedComponentsWithStats(
    img_final, connectivity=8
)

# 2. Coloration des composantes - palette FIXE (générée une fois avec
# np.random.seed(4)) pour que la piste C++ reproduise couleur par couleur sans
# dépendre du générateur numpy. Si n dépasse len(PALETA), cycle.
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]                          # fond noir
img_colored   = colors[labels]
img_annotated = img_colored.copy()

# 3. Tableau de descripteurs
print(f"Composantes détectées (hors fond) : {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", "Segmentation Finale", "Composantes Connexes", "Aires Annotées"],
    cols=4, figsize=(18, 6)
)
```

```
Composantes détectées (hors fond) : 12
```

```
  ID     Área     cx     cy      w      h
-----------------------------------------
   1   231969   1484    280    553    542
   2   158967    917    250    453    468
   3   139229    250    312    433    419
   4   175550    857    821    474    465
   5   222882   1442    889    539    531
   6   210043    316    977    527    516
   7   343376    934   1559    667    665
   8   213641   1555   1574    530    515
   9   147880    328   1543    432    438
  10   187280    363   2111    487    492
  11   150110   1487   2215    433    444
  12   215387    932   2292    528    522
```

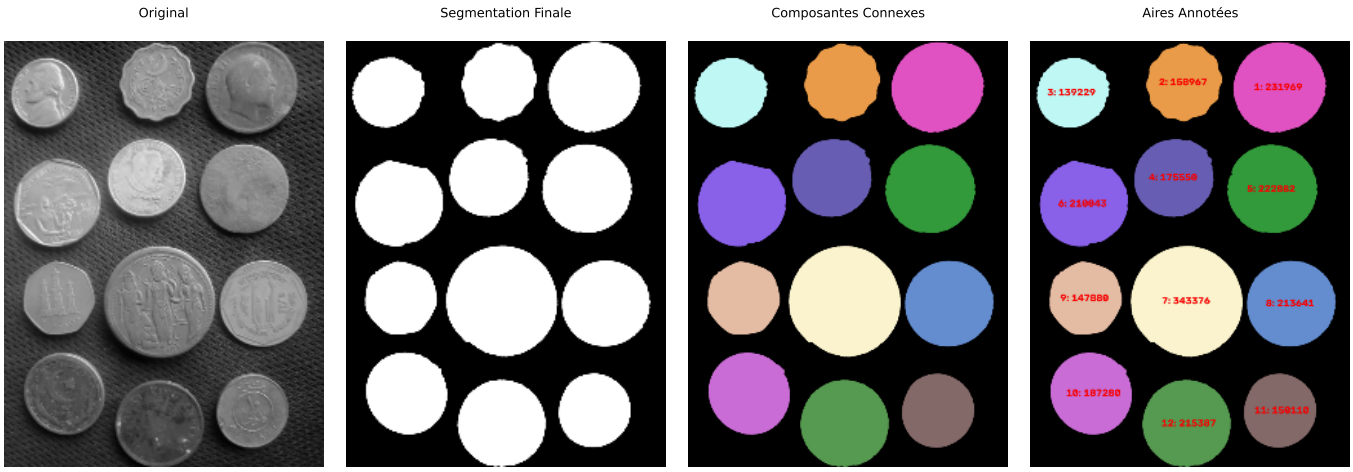


Figure 4.27: Composantes connexes extraites après le *pipeline* CLAHE → Otsu → nettoyage morphologique. Chaque objet est coloré d'une couleur distincte et annoté avec son aire en pixels.

4.5.2 Descripteurs de forme

Alors que `connectedComponentsWithStats` opère sur les régions, l'extraction de contours agit directement sur leurs frontières. L'opérateur retourne, pour chaque objet, une séquence ordonnée de points décrivant son contour.

À partir de cette représentation, il est possible de calculer des descripteurs géométriques qui ne sont pas fournis directement par l'étiquetage des composantes :

- **Aire** (aire du contour)
- **Périmètre** (périmètre) ;
- **Circularité** ($C = \frac{4\pi A}{P^2}$, où A est l'aire et P le périmètre) ;
- **Approximation polygonale** (approximation polygonale) ;
- **Enveloppe convexe** (*convex hull*) ;
- **Hiérarchie des contours**, permettant de représenter les relations parent-enfant entre les régions et leurs trous internes.

La circularité atteint une valeur maximale égale à 1 pour un cercle parfait et diminue à mesure que l'objet devient plus allongé ou présente des irrégularités sur son bord.

La Figure 4.28 présente les contours extraits des pièces de monnaie segmentées, ainsi que les valeurs de circularité calculées pour chaque objet.

```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"Contours détectés : {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=["Segmentation finale", "Contours", "Circularité"],
    cols=3, figsize=(18, 6)
)
```

```
Contours détectés : 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
```

Figure 4.28: Contours extraits avec *findContours*. Chaque pièce est annotée avec sa circularité — des valeurs proches de 1 confirment une forme circulaire.

4.5.3 Connexion avec la détection d'objets moderne

Les descripteurs extraits dans les sections précédentes — notamment les *bounding boxes*, les centroïdes, les aires et les mesures de forme — établissent un pont naturel entre la segmentation morphologique classique et les systèmes modernes de détection d'objets. Bien que les techniques étudiées dans ce chapitre utilisent des opérations sur les pixels et les régions segmentées, bon nombre des représentations produites sont directement compatibles avec les formats employés dans les modèles contemporains de vision par ordinateur.

Les détecteurs basés sur l'apprentissage profond, tels que la famille YOLO (*You Only Look Once*) (REDMON, 2016), opèrent directement sur des images en couleur et produisent, pour chaque objet détecté, une *bounding box* décrite par le centre (cx, cy) et les dimensions (w, h), ainsi qu'une classe et un score de confiance. Cette représentation partage la même structure géométrique de base obtenue par `connectedComponentsWithStats`, bien qu'elle soit produite par un modèle appris et non par une segmentation explicite.

La Figure 4.29 illustre comment les *bounding boxes* obtenues par morphologie peuvent être exportées au format YOLO pour composer des ensembles de données utilisés lors de l'entraînement ou de l'évaluation de détecteurs.

```
# Recalcule les étiquettes/statistiques à partir de la segmentation finale
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 = pièce (unique catégorie dans cet exemple)

print(f"{'cls':>4} {'cx_n':>8} {'cy_n':>8} {'w_n':>8} {'h_n':>8}  ← format 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)

# Exporter le fichier d'annotation au format YOLO
with open("moedas.txt", "w") as f:
    f.write("\n".join(yolo_linhas))
print("\nAnnotation enregistrée dans moedas.txt")

mm.show(
    [img_coins_gray, img_final, img_bbox],
    titles=["Original", "Segmentation Finale", "Boîtes englobantes (format YOLO)"],
    cols=3, figsize=(18, 6)
)
```

```
 cls     cx_n     cy_n      w_n      h_n  ← format 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

Annotation enregistrée dans moedas.txt
```

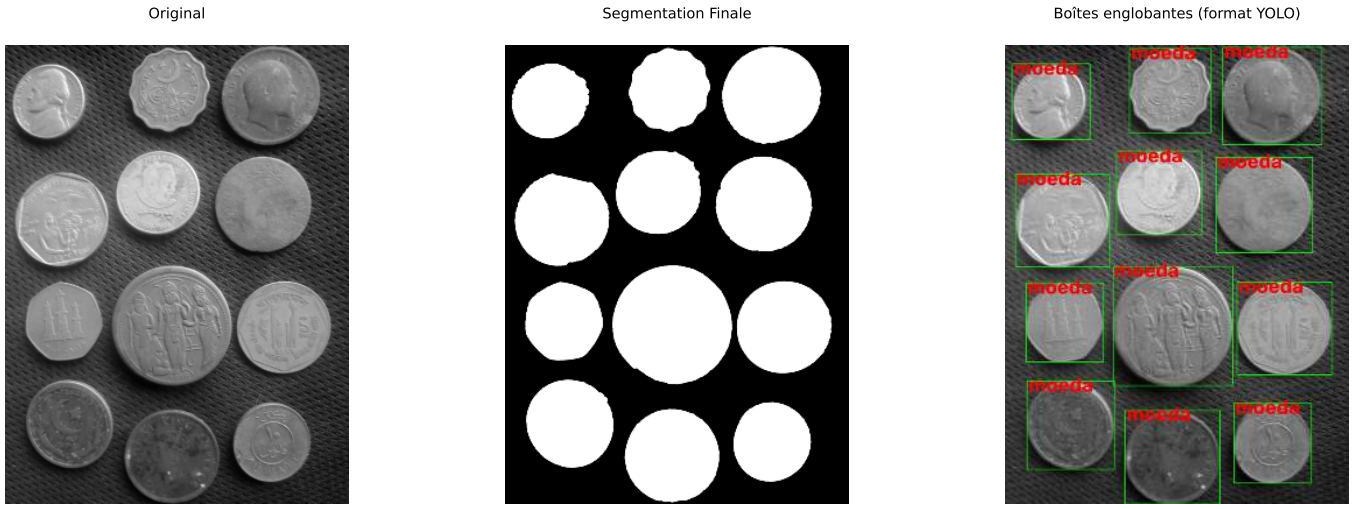


Figure 4.29: *Boîtes englobantes* dérivées des composantes connexes superposées à l'image originale. Les annotations sont exportées au format YOLO (*classe cx cy w h*), avec des coordonnées normalisées pour l'intervalle [0,1].

Le format YOLO stocke chaque objet dans une ligne contenant cinq champs :

```
classe cx cy w h
```

où (cx, cy) représente le centre de la *bounding box* et (w, h) ses dimensions. Toutes les valeurs géométriques sont normalisées dans l'intervalle $[0, 1]$ par rapport à la largeur et à la hauteur de l'image. La **classe** est un identifiant entier associé à une catégorie définie par l'ensemble de données (par exemple, `0 → pièce de monnaie`). Lorsqu'il y a plusieurs catégories — comme la pièce d'or (`0`), la pièce d'argent (`1`) et le disque en plastique (`2`) — il suffit d'attribuer l'identifiant correspondant à chaque objet avant l'exportation, en maintenant exactement le même format d'annotation. Dans l'exemple précédent, ces informations ont été stockées dans le fichier `moedas.txt`.

Le flux présenté dans ce chapitre — segmentation → étiquetage → extraction de *bounding boxes* — correspond conceptuellement à l'étape d'**annotation** (*labeling*) employée dans la construction d'ensembles d'entraînement pour les détecteurs modernes. Des outils spécialisés, comme Label Studio et Roboflow, automatisent ce processus sur des images complexes, mais la logique fondamentale reste la même : associer à chaque objet une région d'intérêt et une classe. Dans des scénarios contrôlés, avec un arrière-plan uniforme et des objets bien séparés, les techniques morphologiques peuvent même générer des annotations automatiquement ou servir de point de départ pour l'étiquetage manuel, réduisant considérablement l'effort de construction de l'ensemble de données. Dans des applications réelles plus complexes, cependant, la validation humaine reste nécessaire pour garantir la qualité des annotations.

Évaluation : IoU (*Intersection over Union*)

Une méthode simple pour évaluer la qualité d'une segmentation consiste à la comparer à un masque de référence (*ground truth*). La métrique la plus couramment utilisée à cette fin est la **IoU** (*Intersection over Union*) :

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} \tag{4.16}$$

où A représente la segmentation produite par l'algorithme et B la segmentation de référence.
La valeur de l'IoU varie entre 0 et 1. Plus la valeur est élevée, plus la superposition entre les masques est importante. Une IoU égale à 1 indique une correspondance parfaite entre la segmentation obtenue et la référence.
La même métrique est également largement utilisée en détection d'objets, où elle est appliquée aux *bounding boxes* prédites et annotées. Dans ce domaine, des valeurs d'IoU supérieures à 0,5 sont souvent adoptées comme critère minimum pour considérer une détection comme correcte.

Au-delà de la morphologie

Les techniques étudiées dans ce chapitre segmentent les objets en exploitant la connectivité spatiale, les opérations morphologiques et le relief topographique. Il existe cependant des approches alternatives basées sur le regroupement de caractéristiques, comme l'algorithme *k-means*, les modèles de mélange gaussien (GMM) et des méthodes plus récentes fondées sur l'apprentissage profond. Ces techniques seront reprises dans la Partie II de l'ouvrage, dédiée à la vision par ordinateur.

4.6 Résumé

Dans ce chapitre, les principales techniques de segmentation et de morphologie mathématique ont été présentées, concluant l'étude du TNI dans le domaine spatial.

- **Prétraitement et seuillage :** La combinaison entre l'égalisation adaptative CLAHE et la méthode d'Otsu s'est révélée efficace pour induire une séparation bimodale dans l'histogramme et simplifier la

binarisation d'images à éclairage non uniforme.

- **Érosion et dilatation :** Opérateurs morphologiques fondamentaux basés sur la recherche de minima et de maxima locaux dans un voisinage défini par l'élément structurant B. Ce sont des opérateurs duaux par complémentation, implémentés au moyen des fonctions `mm.ero` et `mm.dil`.
- **Ouverture et fermeture :** Compositions d'érosion et de dilatation permettant de supprimer les bruits, de lisser les contours et de combler les petites lacunes, tout en préservant la structure globale des objets.
- **Reconstruction morphologique :** Processus géodésique itératif qui propage un marqueur à l'intérieur des limites imposées par un masque, constituant la base d'opérateurs tels que `mm.clohole` et `mm.edgeoff`.
- **Pipeline de nettoyage binaire :** Flux consolidé composé de CLAHE → Otsu → ouverture → `mm.clohole` → ouverture restreinte → `mm.edgeoff`, produisant des masques adaptés à l'analyse quantitative.
- **Morphologie en niveaux de gris :** Extension algébrique fondée sur des minima et maxima pondérés, permettant des opérateurs tels que le gradient morphologique et les filtres *top-hat* pour le rehaussement de structures locales.
- **Transformée de distance et Watershed :** La Transformée de distance a permis de générer des marqueurs automatiques pour l'algorithme *watershed*, rendant possible la séparation d'objets adjacents ou partiellement superposés.
- **Composantes connexes et descripteurs :** L'étiquetage des régions (`mm.label0`) et l'extraction des contours ont permis de calculer des descripteurs géométriques tels que l'aire, le centroïde, le périmètre, la circularité et les *bounding boxes*.
- **Connexion avec la vision par ordinateur moderne :** Les *bounding boxes* extraites par morphologie ont été exportées au format YOLO, mettant en évidence le lien entre les techniques classiques de segmentation et les systèmes modernes de détection d'objets.

Le chapitre 5 introduira les techniques de traitement dans le **domaine fréquentiel**, abordant la **Transformée de Fourier**, le filtrage spectral et les fondements de la **compression d'images**, y compris la DCT, JPEG et les *wavelets*.

4.7 🤖 Utilisation de Gemini Notebook comme tuteur complémentaire

Dans cette édition, l'utilisation de **Gemini Notebook** est encouragée comme outil d'apprentissage complémentaire. Fondé sur l'intelligence artificielle, le système utilise exclusivement les documents fournis par l'auteur comme source de connaissances, produisant des réponses alignées sur le contenu et l'approche adoptés tout au long de ce chapitre.

🎓 Étudiez avec le tuteur intelligent

🚀 ACCÉDER À GEMINI NOTEBOOK : CHAPITRE 04

🌐 **Langue et langage de programmation**

Le projet de ce chapitre dans Gemini Notebook a été construit uniquement avec le texte en **portugais** et les exemples de code en **Python**. Si vous étudiez à partir de l'édition en anglais ou en français, ou si vous suivez le parcours en C++, les réponses du tuteur peuvent ne pas correspondre exactement à la version que vous lisez.

⚠️ **Avertissement concernant le contenu généré par l'IA**

Bien qu'il s'agisse d'un outil d'aide à l'étude précieux, Gemini Notebook peut éventuellement produire des réponses incomplètes, imprécises ou incorrectes. Il est recommandé de valider les informations en consultant le matériel du chapitre, les livres, les articles scientifiques et d'autres sources académiques fiables. Chaque fois que possible, exécutez et expérimentez les exemples pratiques présentés tout au

long du texte afin de consolider la compréhension des concepts.

4.8 Liste d'exercices

1. **(10 %)** Implémentez manuellement le critère d'Otsu sans utiliser `mm.threshold`. Calculez la variance interclasse $\sigma_B^2(T)$ pour tous les seuils $T \in [0, 255]$ en utilisant `mm.hist`, identifiez le seuil optimal T^* et comparez le résultat avec la valeur obtenue par OpenCV. Tracez σ_B^2 en fonction de T et mettez en évidence le point de maximum.

2. **(15 %)** Appliquez un seuillage adaptatif avec des blocs de taille 11, 31 et 51 à une image présentant un éclairage non uniforme. Comparez les résultats avec le seuillage global d'Otsu et discutez des avantages et des limites de chaque approche.

3. **(15 %)** Exécutez le *pipeline watershed* sur l'image de pièces de monnaie en faisant varier le seuil appliqué à la transformée de distance ($0,3$, $0,5$ et $0,7$ fois la valeur maximale). Expliquez comment ce paramètre influence la génération des marqueurs, la séparation des objets adjacents et l'apparition de sur-segmentation.

4. **(15 %)** À l'aide de `mm.drawImg`, construisez une démonstration visuelle pas à pas de l'érosion d'une image binaire 7×7 avec un élément structurant carré de 3×3. Pour chaque position analysée, indiquez si l'élément structurant est entièrement contenu dans l'objet et justifiez la valeur attribuée au pixel de sortie.

5. **(15 %)** Démontrez expérimentalement la dualité entre l'érosion et la dilatation en vérifiant l'identité $(A \ominus B)^c = A^c \oplus \hat{B}$ à l'aide de `mm.ero`, `mm.dil` et `mm.bnot`. Calculez la différence pixel par pixel entre les deux membres de l'équation et présentez le résultat en utilisant `mm.histImg` ou une visualisation équivalente.

6. **(15 %)** Implémentez manuellement le gradient morphologique en utilisant uniquement `mm.ero` et `mm.dil`, en comparant le résultat avec `mm.gradm(img, B)`. Évaluez l'effet de différents éléments structurants (carré 3×3, disque 5×5 et ligne 1×9) sur la détection de contours.

7. **(15 %)** Construisez un *pipeline* complet pour le comptage et la classification de pièces de monnaie par taille (petite, moyenne et grande) en utilisant l'aire et la circularité comme descripteurs. Générez un masque de référence (*ground truth*) manuellement et calculez la métrique IoU (*Intersection over Union*) pour évaluer la qualité de la segmentation. Présentez les résultats dans un tableau et au moyen de visualisations produites avec `mm.show`.

Références du chapitre

La base théorique de ce chapitre s'appuie sur les ouvrages suivants :

- Gonzalez (2018) pour les concepts de segmentation, de seuillage d'Otsu, de transformée de distance, de *watershed*, de morphologie mathématique et de descripteurs de forme.
- Matheron (1975) et Serra (1982) pour la base théorique originale, la formulation algébrique et le développement de la morphologie mathématique.
- Szeliski (2022) pour la segmentation fondée sur les régions, l'étiquetage des composantes connexes, le *watershed* basé sur des marqueurs et l'évaluation de la segmentation au moyen de la métrique IoU.
- Bradski (2008) pour l'utilisation pratique de la bibliothèque OpenCV, y compris des fonctions telles que `mm.dist`, `mm.watershed`, `mm.label0` et l'extraction de contours.
- Redmon (2016) pour une introduction aux détecteurs modernes de la famille YOLO et leur lien avec les descripteurs géométriques comme les *bounding boxes* extraites par segmentation.
- Singh (2024) pour la construction de labyrinthes complexes basés sur des cycles hamiltoniens sur des mosaïques quasi-cristallines, utilisés comme exemple d'application de la transformée de distance géodésique et des algorithmes de recherche de chemins.

- Zampirolli (2025) pour l'implémentation des opérateurs morphologiques, des transformées géodésiques et la résolution de labyrinthes par propagation de distances dans des domaines restreints.

Exécuter Colab | Ouvrir GitHub

Exécuter Colab | Ouvrir GitHub

4.9 💻 Partie Pratique avec Exercices de Programmation

Cette liste transforme les concepts du Chapitre 4 en un parcours pratique de segmentation et de morphologie mathématique. Les exercices de programmation commencent par le seuillage et progressent jusqu'à l'étiquetage et les descripteurs de composants, toujours avec des matrices de petite taille afin que chaque pixel puisse être vérifié à la main.

> ❗ Règle commune des exercices de programmation morphologiques
>
> Dans les opérations avec voisinage, **ne faites pas de padding**. Pour chaque pixel, évaluez uniquement les positions de l'élément structurant qui se trouvent dans le domaine de l'image. C'est la même idée que les implémentations pédagogiques dans `morph.py`, comme `mm.dil0`, `mm.ero0`, `mm.dil1` et `mm.label0` : le voisinage est découpé par le domaine valide de l'image.

🎯 Objectif de ce carnet

Ce carnet permet de développer, valider, organiser et tester des solutions d' **Exercices de Programmation (EP)** dans des environnements interactifs, tels que Colab, avec les mêmes cas de test que Moodle, en y copiant uniquement au moment d'enregistrer la note officielle.

Téléchargement

Téléchargez `morph.py` et `testsuite.py` en exécutant la cellule ci-dessous :

```
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
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0 | TestSuite : 1.1.2
```

Exécution des tests

Pour évaluer les tests, exécutez `TestSuite("EP04_01.extensão").run()` dans une nouvelle cellule, en remplaçant l'extension par celle du langage utilisé (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). Le système télécharge les cas de test depuis GitHub, exécute le programme et calcule automatiquement la note.

Pour tester directement du code Python, sans enregistrer de fichier, utilisez `run_code(codigo)` en passant le code comme *chaîne de caractères* dans une variable `codigo` :

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

4.9.1 EP04_01 Seuillage global par seuil fixe

Dans les **scanners de documents** et les **systèmes de lecture de codes-barres**, la première étape du traitement consiste toujours à séparer ce qui est « objet » (encre, texte, barres) de ce qui est « fond » (papier, emballage). Le **seuillage global** fait exactement cela : il compare chaque pixel à un seuil unique T et décide, en temps réel, s'il appartient à la classe claire ou à la classe sombre. C'est l'opérateur de segmentation le plus simple — et pourtant, il est à l'origine d'une grande partie des *pipelines* industriels d'inspection visuelle. Voir la simulation de cet EP dans Figure 4.30.

4.9.1.1 📋 Directives d'implémentation

1. **Dimensions :** Lire les entiers L (lignes) et C (colonnes).
2. **Seuil :** Lire l'entier T (seuil de décision).
3. **Données :** Lire les valeurs entières de la matrice originale ligne par ligne.
4. **Mappage :** Pour chaque pixel p, calculer la nouvelle valeur à l'aide de l'équation :

$$p' = \begin{cases} 255, & \text{si } p > T \\ 0, & \text{si } p \leq T \end{cases}$$

5. **Sortie :** Afficher la matrice binarisée avec les dimensions $L \times C$.

4.9.1.2 📌 Contraintes computationnelles

- **Binarisation :** La sortie contient **uniquement** les valeurs 0 ou 255.
- **Comparaison stricte :** Le critère utilise $> T$ (les pixels égaux à T deviennent du fond).
- **Type :** Le résultat final doit être entier.
- **Remarque :** Cet EP suit la convention d'OpenCV (`cv2.THRESL_BINARY`) : seuls les pixels avec une valeur **supérieure à** T deviennent blancs (`255`) ; les pixels avec une valeur **égale à** T restent noirs (`0`).

4.9.1.3 🧠 Fondement théorique

Paramètre	Type	Impact visuel
T **petit**	Entier	La plupart des pixels deviennent blancs
T **grand**	Entier	La plupart des pixels deviennent noirs
T **bien choisi**	Entier	Sépare nettement l'objet et le fond

4.9.1.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Entier T.
- Lignes suivantes : Éléments entiers de la matrice originale.

Sortie :

- Matrice binarisée en L lignes et C colonnes, valeurs 0 ou 255 séparées par des espaces.

4.9.1.5 📌 Exemples

Entrée	Sortie	Remarque
2 4 100 0 99 100 180 255 30 120 80	0 0 0 255 255 0 255 0	$T = 100$: seuls les pixels avec une valeur supérieure à 100 deviennent blancs ; par conséquent, 99 et 100 deviennent noirs.
1 3 0 0 50 255	0 255 255	$T = 0$: seuls les pixels avec une valeur strictement supérieure à 0 deviennent blancs.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-ep0401-limiar.html

Figure 4.30: Simulateur EP04_01 : Seuillage global par seuil fixe (p' = (p > T) ? 255 : 0)

```
%%writefile EP04_01.py
# Code Python
```

```
Overwriting EP04_01.py
```

```
TestSuite("EP04_01.py").run()
```

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

4.9.2 EP04_02 📊 Seuillage automatique d'Otsu

Choisir manuellement le seuil T fonctionne lorsque l'éclairage est stable, mais en **microscopie numérique** et en **inspection de lames de sang**, chaque échantillon présente un contraste différent — un seuil fixe échouerait d'une image à l'autre. La **méthode d'Otsu** résout ce problème en trouvant, de manière autonome,

le seuil qui **maximise la séparation statistique** entre les deux classes de pixels, rendant la segmentation automatique et adaptative. Voir dans Figure 4.31 une simulation de cet EP.

4.9.2.1 📋 Directives d'implémentation

1. **Dimensions :** Lire les entiers L (lignes) et C (colonnes).
2. **Données :** Lire les valeurs entières de la matrice originale ligne par ligne.
3. **Histogramme :** Construire l'histogramme $h[i]$, $i = 0, \dots, 255$, en comptant combien de pixels ont la valeur i.
4. **Recherche du seuil :** Pour chaque candidat T de 1 à 255, calculer la **variance inter-classes** :

$$\sigma_B^2(T) = \frac{n_0 \cdot n_1}{N^2} (m_0 - m_1)^2$$

   où n_0, n_1 sont les quantités de pixels ayant une valeur $< T$ et $\geq T$, m_0, m_1 sont leurs moyennes, et $N = L \times C$.
5. **Choix :** Le seuil optimal T^* est celui qui maximise $\sigma_B^2(T)$ (en cas d'égalité, conserver le **premier** trouvé).
6. **Application :** Binariser l'image en utilisant T^*, en appliquant :

$$p' = \begin{cases} 255, & \text{si } p > T^* \\ 0, & \text{si } p \leq T^* \end{cases}$$

4.9.2.2 📌 Contraintes computationnelles

- **Candidats valides :** Ignorer T qui laisse $n_0 = 0$ ou $n_1 = 0$ (classe vide).
- **Égalité :** Toujours conserver le **premier** T qui a atteint la valeur maximale de σ_B^2.
- **Type :** T^* et la matrice de sortie doivent être des entiers.
- **Convention OpenCV :** La binarisation suit `cv2.THRESL_BINARY` ; les pixels ayant une valeur exactement égale à T^* deviennent noirs.

4.9.2.3 🧠 Fondements théoriques

Concept	Signification	Impact
$\sigma_B^2(T)$ **élevée**	Classes bien séparées en T	T est un bon candidat comme seuil
Histogramme bimodal	Deux « pics » distincts	Otsu trouve le creux entre eux
Histogramme unimodal	Un seul « pic »	Otsu choisit toujours *un* T, mais la segmentation est peu fiable

4.9.2.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

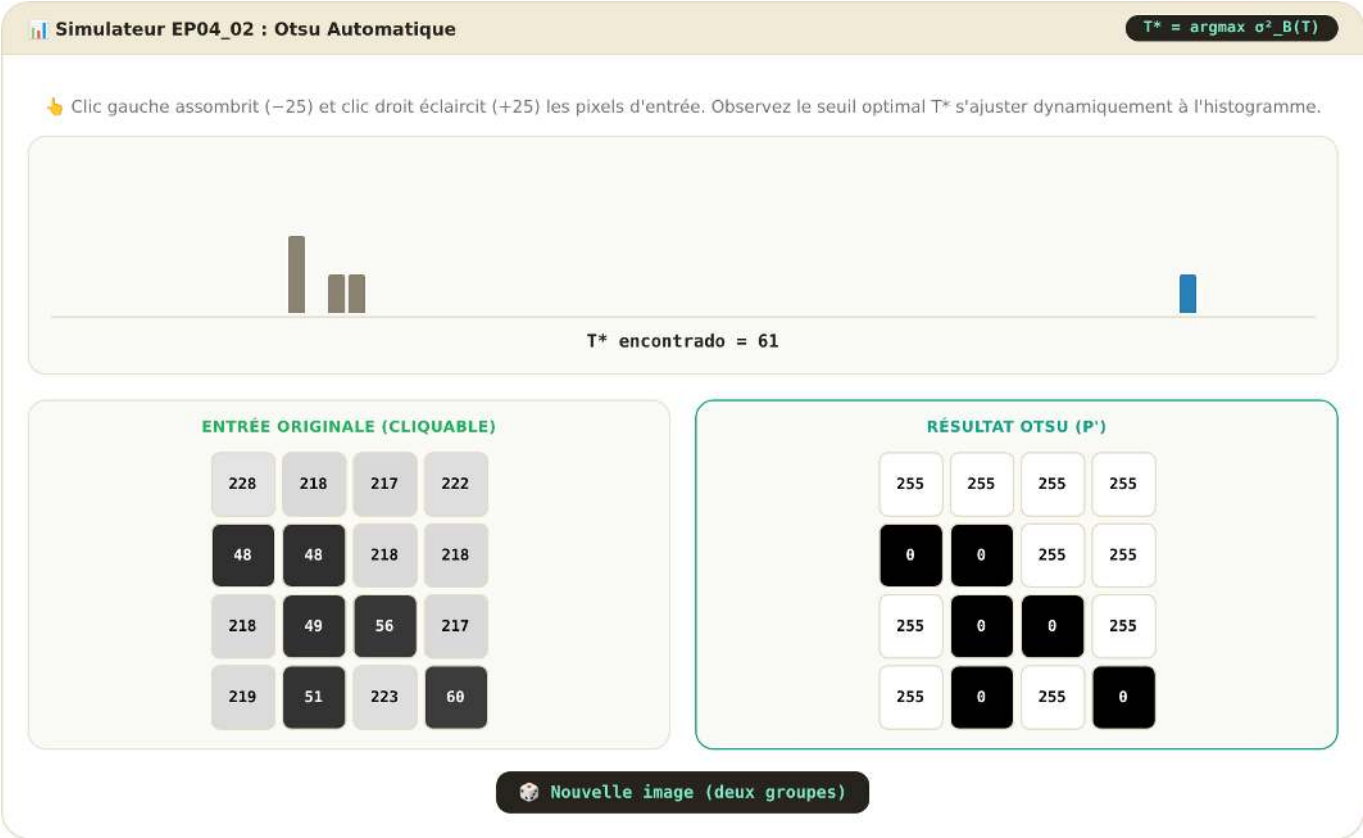
- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Lignes suivantes : Éléments entiers de la matrice originale.

Sortie :

- Matrice binarisée en L lignes et C colonnes, valeurs 0 ou 255.

4.9.2.5 📌 Exemples

Entrée	Sortie	Observation
4 4 12 12 12 200 12 12 200 200 12 200 200 200 200 200 200 200	0 0 0 255 0 0 255 255 0 255 255 255 255 255 255 255	Histogramme bimodal net : 12 et 200
1 2 10 250	0 250	Deux valeurs seulement : T^* reste sur la plus grande



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-ep0402-otsu.html

Figure 4.31: Simulateur EP04_02 : Seuillage automatique d'Otsu (T* = argmax ²_B(T))

```
%%writefile EP04_02.py
# Code Python
```

```
Overwriting EP04_02.py
```

```
TestSuite("EP04_02.py").run()
```

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

4.9.3 EP04_03 🌱 Dilatation binaire plane (mm.dil0)

En **microscopie de particules** et en **OCR de plaques d'immatriculation usées**, les traits fins ou discontinus doivent être « épaissis » pour que la reconnaissance fonctionne. La **dilatation morphologique**

fait exactement cela : elle étend les régions claires à l'aide d'un élément structurant B — la même opération implémentée dans `morph.py` comme `mm.dilO(f, B)`, utilisée lorsque B est **plat** (sans poids, uniquement 0/1). Voir dans Figure 4.32 une simulation de cet EP.

4.9.3.1 📋 Directives d'implémentation

1. **Dimensions de l'image :** Lire les entiers L (lignes) et C (colonnes) de f.
2. **Dimensions de B :** Lire les entiers L_B (lignes) et C_B (colonnes) de l'élément structurant.
3. **Élément structurant :** Lire la matrice B avec des valeurs 0 ou 1, ligne par ligne.
4. **Données :** Lire la matrice f (l'image originale), ligne par ligne.
5. **Réflexion :** Construire B_{ref}, la version de B réfléchie à 180° (lignes et colonnes inversées) — exactement comme le fait `mm.dilO` en interne.
6. **Voisinage sans padding :** Pour chaque pixel (y, x), parcourir les positions (by, bx) de B_{ref} centrées en (y, x), en utilisant le décalage

$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\tfrac{L_B}{2} + 0{,}5, \quad o_x = -\tfrac{C_B}{2} + 0{,}5$$

   Écarter tout (v_y, v_x) hors de $[0, L) \times [0, C)$ — **ne pas remplir avec des zéros**.
7. **Mappage :** Calculer chaque pixel de sortie comme le **maximum** entre $f(y, x)$ et tous les $f(v_y, v_x)$ valides dont la position correspondante dans B_{ref} vaut 1 :

$$g(y, x) = \max\Big(f(y, x), \max_{\substack{(v_y, v_x) \text{ valide} \\ B_{ref}(by, bx)=1}} f(v_y, v_x)\Big)$$

8. **Sortie :** Afficher la matrice g avec les dimensions $L \times C$.

4.9.3.2 📌 Contraintes computationnelles

- **Sans padding :** Ne jamais inventer de voisins hors de l'image ; n'utiliser que ceux qui existent réellement.
- **Réflexion obligatoire :** B doit être réfléchi avant d'être appliqué (c'est ce qui distingue `mm.dilO` d'une simple recherche de maximum).
- **Robustesse des bords :** Si aucune position valide de $B_{ref} = 1$ ne tombe dans le domaine pour un pixel donné, celui-ci **conserve sa valeur originale**.

4.9.3.3 🧠 Fondement théorique

Concept	Signification	Impact visuel
Dilatation	$g \geq f$ toujours (extensive)	Les régions claires croissent, les trous sombres rétrécissent
B plus grand	Voisinage plus large	Croissance plus agressive
Réflexion de B	$B_{ref}(y, x) = B(-y, -x)$	Garantit la définition formelle de Minkowski de la dilatation

4.9.3.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : entier L.
- Ligne 2 : entier C.
- Ligne 3 : entier L_B.
- Ligne 4 : entier C_B.
- Les L_B lignes suivantes : éléments entiers (0 ou 1) de la matrice B.
- Les L lignes suivantes : éléments entiers de la matrice f.

Sortie :

- Matrice g en L lignes et C colonnes, valeurs entières séparées par des espaces.

4.9.3.5 📌 Exemples

Entrée	Sortie	Observation
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 croix symétrique : point isolé se dilate en croix
1 4 1 3 1 1 1 10 200 5 80	200 200 200 80	B horizontal : chaque pixel « attire » le maximum des voisins de la ligne

```
%%writefile EP04_03.py
# Code Python
```

```
Overwriting EP04_03.py
```

```
TestSuite("EP04_03.py").run()
```

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

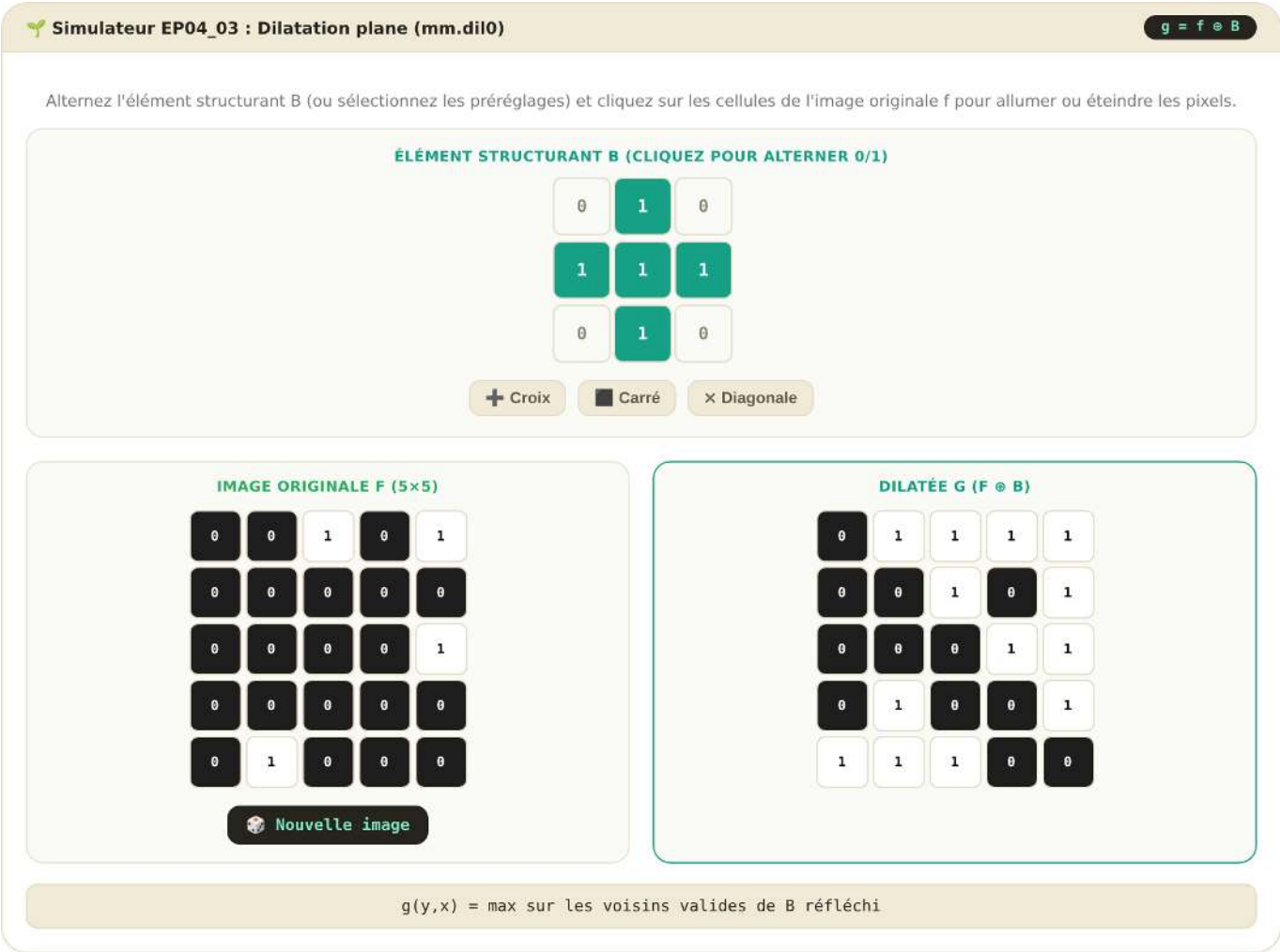
4.9.4 EP04_04 🪨 Érosion binaire plane (mm.ero0)

Si la dilatation épaissit, l'**érosion** affine. Dans les **systèmes de comptage de cellules**, elle est utilisée pour **séparer les cellules qui se touchent** : en « mangeant » les bords de chaque région, les connexions fines entre objets disparaissent avant même qu'un comptage soit effectué. Dans `morph.py`, c'est l'opération `mm.ero0(f, B)` — le **dual** exact de la dilatation, et la seule des deux qui **ne** réfléchit pas l'élément structurant. Voir dans Figure 4.33 une simulation de cet EP.

4.9.4.1 📋 Directives d'implémentation

1. **Dimensions de l'image :** Lire les entiers L (lignes) et C (colonnes) de f.
2. **Dimensions de B :** Lire les entiers L_B (lignes) et C_B (colonnes) de l'élément structurant.
3. **Élément structurant :** Lire la matrice B avec des valeurs 0 ou 1, ligne par ligne.
4. **Données :** Lire la matrice f (l'image originale), ligne par ligne.
5. **Voisinage sans padding (sans réflexion !) :** Pour chaque pixel (y, x), parcourir les positions (by, bx) de B **dans l'ordre original** (sans réfléchir), en utilisant le même décalage que dans l'EP04_03 :

$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\frac{L_B}{2} + 0{,}5, \quad o_x = -\frac{C_B}{2} + 0{,}5$$



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-ep0403-dilatacao.html

Figure 4.32: Simulateur EP04_03 : Dilatation binaire plane (g = f B)

Écarter tout (v_y, v_x) hors de $[0, L) \times [0, C)$. 6. **Mappage :** Calculer chaque pixel de sortie comme le **minimum** entre $f(y, x)$ et tous les $f(v_y, v_x)$ valides dont la position correspondante dans B vaut 1 :

$$g(y, x) = \min \Big(f(y, x), \min_{\substack{(v_y, v_x) \text{ valide} \\ B(by, bx)=1}} f(v_y, v_x) \Big)$$

7. **Sortie :** Afficher la matrice g avec des dimensions $L \times C$.

4.9.4.2 📌 Contraintes de calcul

- **Sans réflexion :** Contrairement à la dilatation, B est utilisé **exactement comme lu** — réfléchir ici serait une erreur conceptuelle grave.
- **Sans padding :** Les voisins hors de l'image sont simplement ignorés, jamais traités comme 0.
- **Robustesse de bord :** Si aucune position valide de $B = 1$ ne tombe dans le domaine, le pixel conserve sa valeur originale.

4.9.4.3 🧠 Fondements théoriques

Concept	Signification	Impact visuel
Érosion	$g \leq f$ toujours (anti-extensive)	Les régions claires rétrécissent, le bruit ponctuel disparaît
Dualité	$\text{ero}(f, B) = -\text{dil}(-f, B_{ref})$	Érosion et dilatation sont des « miroirs » mathématiques
B **plus grand**	Érosion plus agressive	Les objets fins disparaissent complètement

4.9.4.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Entier L_B.
- Ligne 4 : Entier C_B.
- L_B lignes suivantes : éléments entiers (0 ou 1) de la matrice B.
- L lignes suivantes : éléments entiers de la matrice f.

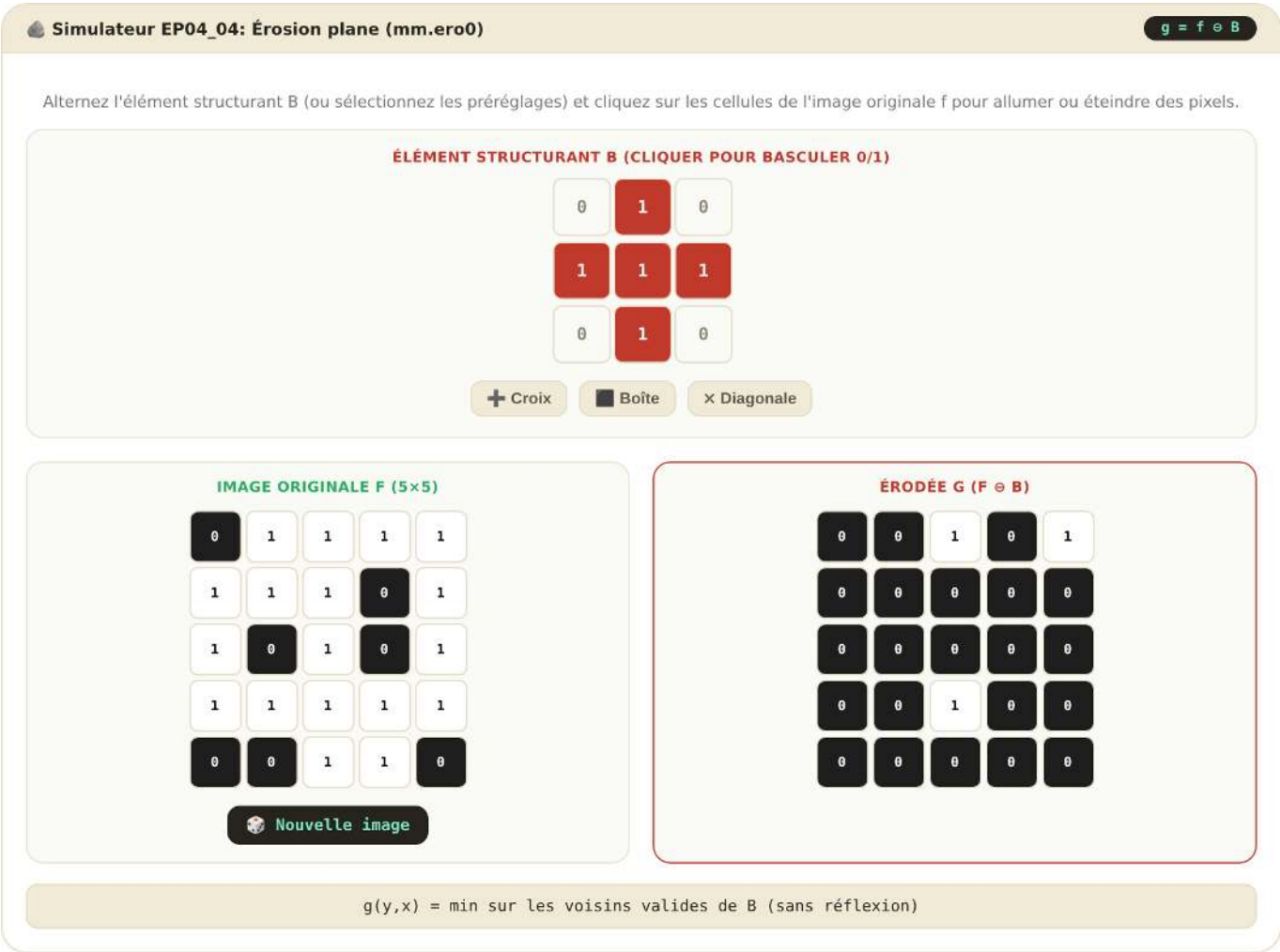
Sortie :

- Matrice g en L lignes et C colonnes, valeurs entières séparées par des espaces.

4.9.4.5 📌 Exemples

Entrée	Sortie	Observation
3 3 3 3 0 1 0 1 1 1 0 1 0 9 9 9 9 0 9 9 9 9	9 0 9 0 0 0 9 0 9	Le « trou » central (0) se propage en croix

Entrée	Sortie	Observation
1 4 1 3 1 1 1 10 200 5 80	10 5 5 80	B horizontal : chaque pixel « tire » le minimum des voisins de la ligne



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-ep0404-erosao.html

Figure 4.33: Simulateur EP04_04: Érosion Binaire Plane (g = f B)

```
%%writefile EP04_04.py
# Code Python
```

```
Overwriting EP04_04.py
```

```
TestSuite("EP04_04.py").run()
```

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

4.9.5 EP04_05 🧹 Ouverture Morphologique (Suppression de Bruit)

Les images capturées par des **capteurs à faible coût**, comme ceux des drones agricoles, sont souvent parsemées de petits points de bruit — des pixels isolés qui ne représentent rien de réel. Appliquer une érosion

suivie d'une dilatation avec le **même** élément structurant produit l'**ouverture** : elle « nettoie » les points et les fines protubérances, mais rend à l'objet principal pratiquement sa taille d'origine. C'est la combinaison classique utilisée dans le **pré-traitement d'images satellitaires** avant tout comptage de surface cultivée. Voir dans Figure 4.34 une simulation de cet EP.

4.9.5.1 📋 Directives d'Implémentation

1. **Dimensions de l'image :** Lire les entiers L (lignes) et C (colonnes) de f.
2. **Dimensions de B :** Lire les entiers L_B (lignes) et C_B (colonnes) de l'élément structurant.
3. **Élément structurant :** Lire la matrice B avec des valeurs 0 ou 1, ligne par ligne.
4. **Données :** Lire la matrice binaire f (valeurs 0 ou 1), ligne par ligne.
5. **Érosion :** Calculer $e = f \ominus B$, en utilisant exactement l'algorithme de l'EP04_04 (sans réfléchir B, sans padding).
6. **Dilatation :** Calculer $g = e \oplus B$, en utilisant exactement l'algorithme de l'EP04_03 (en réfléchissant B, sans padding) — mais maintenant appliqué sur e, pas sur f.
7. **Sortie :** Afficher la matrice résultante g (l'**ouverture** de f par B) avec les dimensions $L \times C$.

4.9.5.2 📌 Contraintes Computationnelles

- **Ordre fixe :** C'est **toujours** l'érosion d'abord, puis la dilatation — l'ordre inverse définit un autre opérateur (la fermeture, du prochain EP).
- **Même B :** L'élément structurant utilisé pour l'érosion et la dilatation doit être identique.
- **Sans padding dans aucune des deux étapes.**

4.9.5.3 🧠 Fondement Théorique

Concept	Signification	Impact Visuel
Anti-extensivité	$g \subseteq f$ toujours	L'ouverture ne crée jamais de nouveau pixel, elle ne fait que supprimer
Idempotence	ouverture(ouverture(f)) = ouverture(f)	Réappliquer ne change plus rien
Points isolés	Plus petits que B	Ils sont complètement éliminés
Noyau de l'objet	Plus grand que B	Il est récupéré presque intact par la dilatation finale

4.9.5.4 📦 Spécification d'Entrée et de Sortie (VPL)

Entrée :

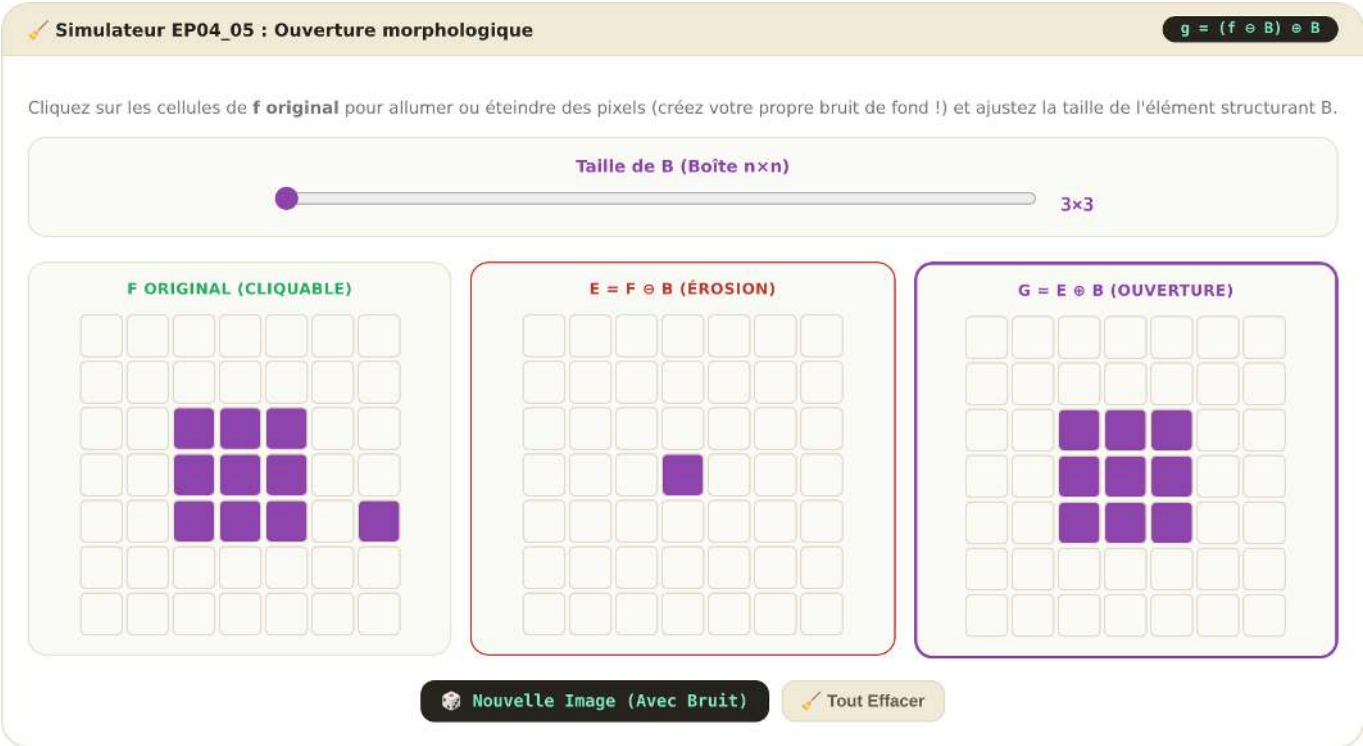
- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Entier L_B.
- Ligne 4 : Entier C_B.
- Les L_B lignes suivantes : éléments entiers (0 ou 1) de la matrice B.
- Les L lignes suivantes : éléments entiers (0 ou 1) de la matrice f.

Sortie :

- Matrice résultante en L lignes et C colonnes, valeurs 0 ou 1.

4.9.5.5 📌 Exemples

Entrée	Sortie	Observation
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	Les points isolés et la fine protubérance disparaissent ; le carré central survit



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-ep0405-abertura.html

Figure 4.34: Simulateur EP04_05 : Ouverture morphologique (g = (f B) B)

```
%%writefile EP04_05.py
# Code Python
```

```
Overwriting EP04_05.py
```

```
TestSuite("EP04_05.py").run()
```

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

4.9.6 EP04_06 🧩 Fermeture Morphologique (Remplissage des Lacunes)

Dans la **numérisation d'empreintes digitales**, les sillons de la peau sont parfois interrompus par de la saleté ou un dessèchement, créant de petites lacunes dans la courbe continue qui devrait exister. La **fermeture** — dilatation suivie d'une érosion avec le même élément structurant — est l'opérateur dual de l'ouverture : elle **remplit les petits trous et les renfoncements étroits**, sans modifier significativement le contour externe de l'objet. C'est l'étape standard avant d'extraire le squelette d'une empreinte digitale. Voir dans Figure 4.35 une simulation de cet EP.

4.9.6.1 📋 Directives d'implémentation

1. **Dimensions de l'image :** Lire les entiers L (lignes) et C (colonnes) de f.
2. **Dimensions de B :** Lire les entiers L_B (lignes) et C_B (colonnes) de l'élément structurant.
3. **Élément structurant :** Lire la matrice B avec les valeurs 0 ou 1, ligne par ligne.
4. **Données :** Lire la matrice binaire f (valeurs 0 ou 1), ligne par ligne.
5. **Dilatation :** Calculer $d = f \oplus B$, en utilisant exactement l'algorithme de l'EP04_03 (réflexion de B, sans padding).
6. **Érosion :** Calculer $g = d \ominus B$, en utilisant exactement l'algorithme de l'EP04_04 (sans réflexion de B, sans padding) — maintenant appliqué sur d, et non sur f.
7. **Sortie :** Afficher la matrice résultante g (la **fermeture** de f par B) avec les dimensions $L \times C$.

4.9.6.2 📌 Contraintes de calcul

- **Ordre fixe :** C'est **toujours** la dilatation d'abord, puis l'érosion — l'ordre inverse est l'ouverture de l'EP04_05.
- **Même B :** L'élément structurant utilisé dans la dilatation et dans l'érosion doit être identique.
- **Sans padding à aucune des deux étapes.**

4.9.6.3 🧠 Fondement théorique

Concept	Signification	Impact visuel
Extensivité	$g \supseteq f$ toujours	La fermeture ne supprime jamais de pixel, elle n'ajoute que
Idempotence	$\text{fermeture}(\text{fermeture}(f)) = \text{fermeture}(f)$	Réapplique ne change plus rien
Petits trous	Plus petits que B	Sont complètement remplis
Dualité	$\text{fermeture}(f) = \overline{\text{ouverture}(\bar{f})}$	C'est l'ouverture appliquée au « négatif » de l'image

4.9.6.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

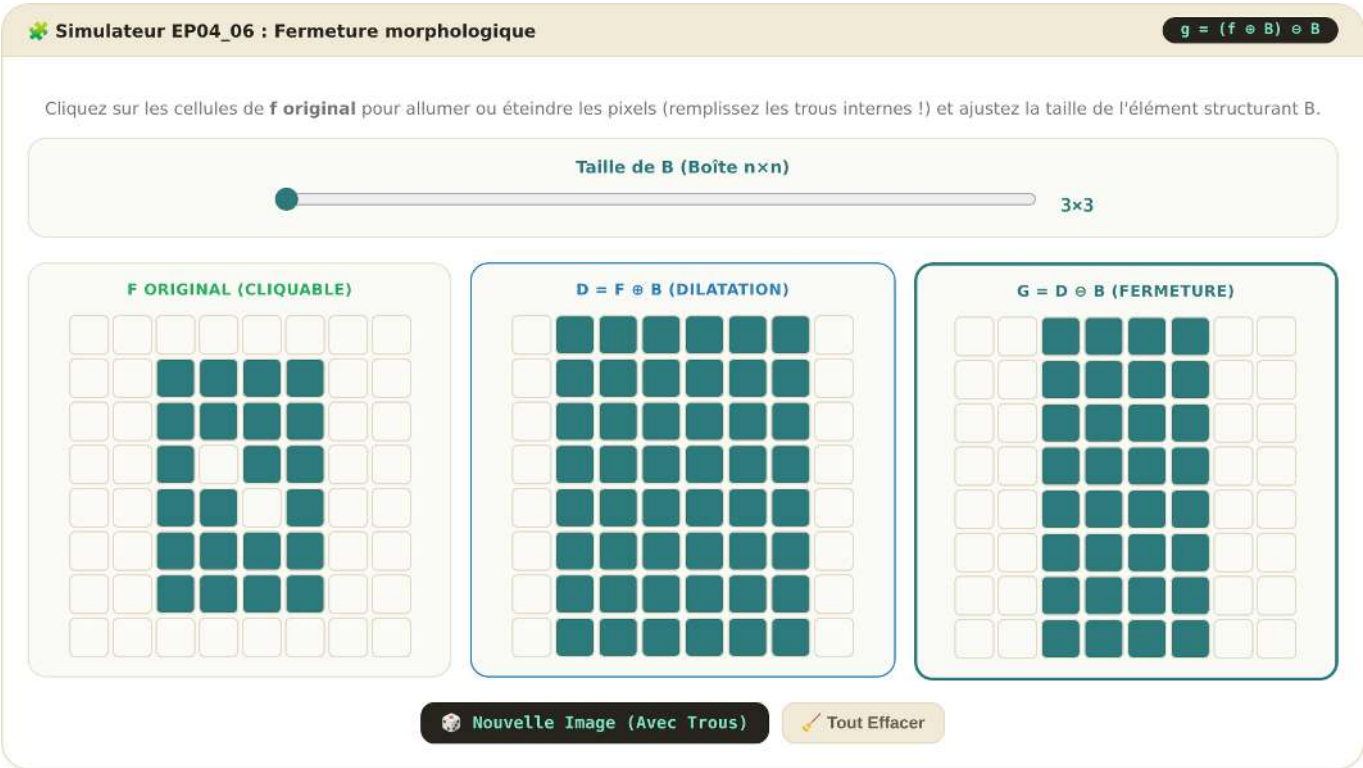
- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Entier L_B.
- Ligne 4 : Entier C_B.
- Les L_B lignes suivantes : éléments entiers (0 ou 1) de la matrice B.
- Les L lignes suivantes : éléments entiers (0 ou 1) de la matrice f.

Sortie :

- Matrice résultante en L lignes et C colonnes, valeurs 0 ou 1.

4.9.6.5 📌 Exemples

Entrée	Sortie	Observation
8 8 3 3 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0	0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0	Les deux trous internes non adjacents sont totalement remplis



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-ep0406-fechamento.html

Figure 4.35: Simulateur EP04_06: Fermeture morphologique (g = (f B) B)

```
%%writefile EP04_06.py
# Code Python
```

```
Overwriting EP04_06.py
```

```
TestSuite("EP04_06.py").run()
```

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

4.9.7 EP04_07 Dilatation et Érosion Pondérées (mm.dil1 / mm.ero1)

Jusqu'à présent, l'élément structurant disait simplement « ce voisin compte » ou « ne compte pas » — mais dans les **modèles numériques de terrain** (utilisés en SIG et en planification du drainage urbain), chaque voisin devrait avoir un **poids différent** selon la distance ou la direction du relief. Les versions **pondérées** de la dilatation et de l'érosion, implémentées dans `morph.py` comme `mm.dil1(f, b)` et `mm.ero1(f, b)`, additionnent (ou soustraient) le poids de chaque voisin avant de prendre le maximum (ou le minimum) — généralisant ainsi tout ce qui a été fait dans les EP précédents. Voir dans Figure 4.36 une simulation de cet EP.

4.9.7.1 📋 Directives d'Implémentation

1. **Dimensions de l'image :** Lire les entiers L (lignes) et C (colonnes) de f.
2. **Dimensions de b :** Lire les entiers L_B (lignes) et C_B (colonnes) de l'élément structurant pondéré.
3. **Poids :** Lire la matrice b des poids **entiers** (ils peuvent être négatifs, nuls ou positifs), ligne par ligne.
4. **Données :** Lire la matrice f (l'image d'origine), ligne par ligne.
5. **Voisinage sans padding :** Pour chaque pixel (y, x), parcourir **toutes** les positions (by, bx) de b (pas seulement celles où la valeur serait 1 — ici **tout** poids participe), en utilisant le même décalage que dans les EP précédents :

$$v_y = y + by + o_y, \quad v_x = x + bx + o_x, \quad o_y = -\tfrac{L_B}{2} + 0{,}5, \quad o_x = -\tfrac{C_B}{2} + 0{,}5$$

   Écarter tout (v_y, v_x) hors de $[0, L) \times [0, C)$.
6. **Dilatation pondérée :** Calculer

$$g_{dil}(y, x) = \max\Big(f(y, x), \max_{(v_y, v_x)\ \text{valide}} \big(f(v_y, v_x) + b(by, bx)\big)\Big)$$

7. **Érosion pondérée :** Calculer, **en utilisant le même b et sans réflexion** :

$$g_{ero}(y, x) = \min\Big(f(y, x), \min_{(v_y, v_x)\ \text{valide}} \big(f(v_y, v_x) - b(by, bx)\big)\Big)$$

8. **Sortie :** Afficher **d'abord** la matrice complète g_{dil}, puis **ensuite** la matrice complète g_{ero}.

4.9.7.2 📌 Contraintes Computationnelles

- **Aucune des deux ne réfléchit** b — la version pondérée n'utilise pas la réflexion, même pour la dilatation (contrairement à `mm.dil0`).
- **Tous les poids participent :** Il n'existe pas ici de filtre « $B = 1$ » ; même un poids 0 entre en compte.
- **Sans padding :** les voisins hors de l'image sont ignorés, jamais virtuellement remplis.
- **Type :** La sortie peut contenir des valeurs négatives ou supérieures à 255 — **pas** de *clipping* dans cet EP.
- **Astuce :** Pour supprimer les messages de dépassement lors du dépassement des limites du type uint8, inclure au début du code :

```
import warnings
warnings.filterwarnings("ignore")
```

4.9.7.3 🧠 Fondement Théorique

Concept	Signification	Impact Visuel
Poids positif	« Tire » la valeur du voisin vers le haut lors de la dilatation	Simule un relief qui monte dans cette direction
Poids négatif	Réduit la contribution du voisin	Simule la distance ou une atténuation directionnelle

Concept	Signification	Impact Visuel
Dualité pondérée	$\text{ero1}(f, b) = -\text{dil1}(-f, b)$	La symétrie entre les deux opérations se maintient même avec des poids

4.9.7.4 📦 Spécification d'Entrée et de Sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Entier L_B.
- Ligne 4 : Entier C_B.
- Les L_B lignes suivantes : éléments entiers (pouvant être négatifs) de la matrice b.
- Les L lignes suivantes : éléments entiers de la matrice f.

Sortie :

- D'abord la matrice g_{dil} en L lignes et C colonnes.
- Ensuite la matrice g_{ero} en L lignes et C colonnes.

4.9.7.5 📌 Exemples

Entrée	Sortie	Observation
3 3 3 3 0 1 0 1 2 1 0 1 0 10 20 30 40 50 60 70 80 90	50 60 61 80 90 91 81 91 92 8 9 19 9 10 20 39 40 50	Le poids central 2 accélère la croissance lors de la dilatation et le rétrécissement lors de l'érosion

```
%%writefile EP04_07.py
# Code Python
```

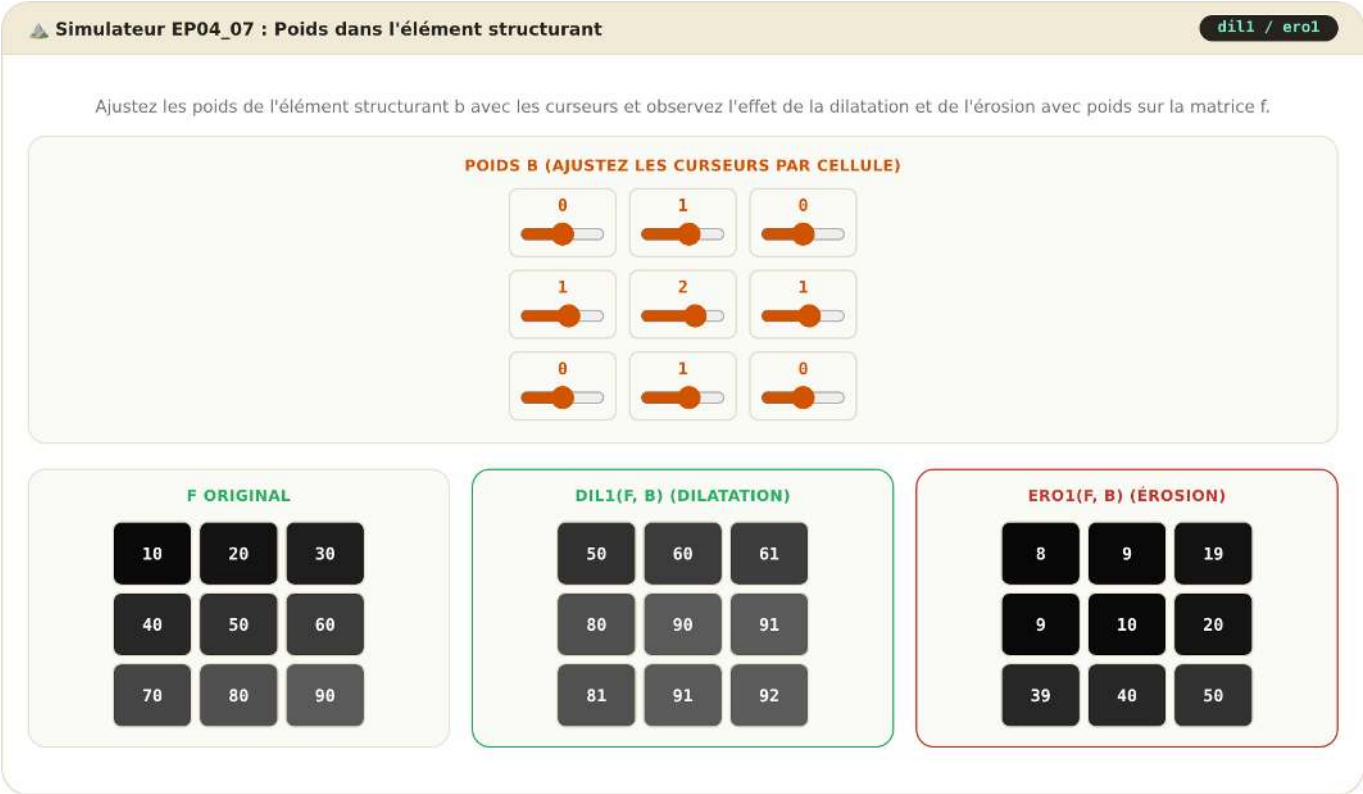
```
Overwriting EP04_07.py
```

```
TestSuite("EP04_07.py").run()
```

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

4.9.8 EP04_08 🎉 Gradient morphologique, Top-hat et Black-hat

En **inspection automatique de plaques de circuits**, trois questions reviennent constamment : où se trouvent les **bords** des composants ? Quels **détails clairs et petits** (comme les points de soudure) se détachent du fond ? Quelles **cavités sombres** (comme les fissures) le fond dissimule-t-il ? Une seule paire érosion/dilatation répond aux trois : le **gradient morphologique** met en évidence les contours, le **top-hat** révèle les pics étroits, et le **black-hat** révèle les vallées étroites — trois outils, un seul voisinage. Voir dans Figure 4.37 une simulation de cet EP.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-ep0407-pesos.html

Figure 4.36: Simulateur EP04_07 : Dilatation et Érosion avec Poids (mm.dil1 / mm.ero1)

4.9.8.1 📋 Directives d'implémentation

1. **Dimensions de l'image :** Lire les entiers L (lignes) et C (colonnes) de f.
2. **Dimensions de B :** Lire les entiers L_B (lignes) et C_B (colonnes) de l'élément structurant.
3. **Élément structurant :** Lire la matrice B avec des valeurs 0 ou 1, ligne par ligne.
4. **Données :** Lire la matrice f (l'image originale, en niveaux de gris), ligne par ligne.
5. **Opérateurs de base :** Calculer, exactement comme dans les EP 04_03 à 04_06 :
   - $d = f \oplus B$ (dilatation),
   - $e = f \ominus B$ (érosion),
   - ouverture $= e \oplus B$,
   - fermeture $= d \ominus B$.
6. **Gradient morphologique :** $\text{grad}(y, x) = d(y, x) - e(y, x)$.
7. **Top-hat :** $\text{tophat}(y, x) = f(y, x) - \text{ouverture}(y, x)$.
8. **Black-hat :** $\text{blackhat}(y, x) = \text{fermeture}(y, x) - f(y, x)$.
9. **Sortie :** Afficher, **dans cet ordre**, les trois matrices complètes : gradient, top-hat, black-hat.

4.9.8.2 📌 Contraintes computationnelles

- **Sans padding à aucune étape intermédiaire** — dilatation, érosion, ouverture et fermeture suivent les mêmes règles de voisinage que les EP précédents.
- **Pas de *clipping* :** les trois sorties peuvent contenir n'importe quelle valeur entière (le gradient est toujours ≥ 0, mais top-hat et black-hat le sont aussi).
- **Réutilisation :** d et e doivent être calculés **une seule fois** et réutilisés pour construire ouverture, fermeture et gradient.

4.9.8.3 🧠 Fondements théoriques

Opérateur	Formule	Ce qu'il révèle
Gradient	$d - e$	Bords : zéro dans les régions planes, élevé aux transitions
Top-hat	$f - \text{ouverture}(f)$	Éléments **clairs et fins**, plus petits que B
Black-hat	$\text{fermeture}(f) - f$	Éléments **sombres et fins**, plus petits que B

4.9.8.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Entier L_B.
- Ligne 4 : Entier C_B.
- Lignes suivantes L_B : éléments entiers (0 ou 1) de la matrice B.
- Lignes suivantes L : éléments entiers de la matrice f.

Sortie :

- Matrice gradient en L lignes et C colonnes.
- Matrice top-hat en L lignes et C colonnes.
- Matrice black-hat en L lignes et C colonnes.

4.9.8.5 📌 Exemples

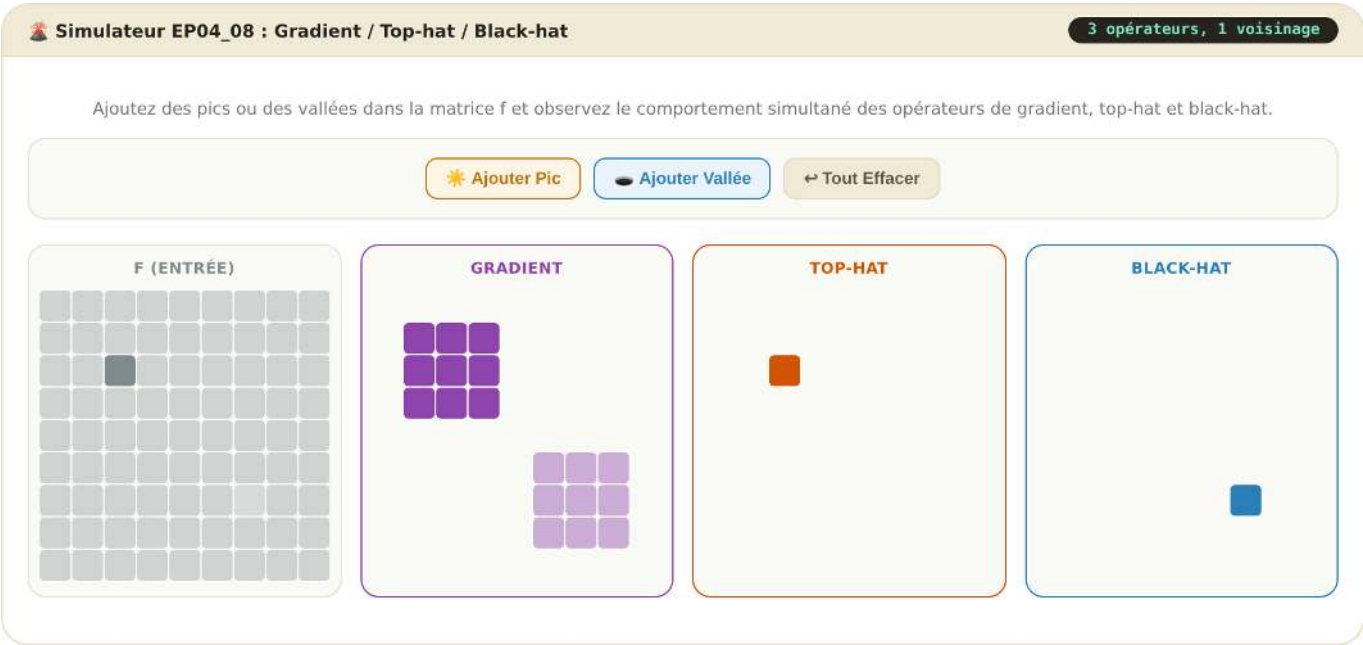
Entrée	Sortie	Observation
9 9 3 3 1 1 1 1 1 1 1 1 1 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 80 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 2 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10	(gradient : halo $3 \times 3 = 70$ autour de $(2, 2)$ et halo $3 \times 3 = 8$ autour de $(6, 6)$, reste 0) (top-hat : unique 70 en $(2, 2)$, reste 0) (black-hat : unique 8 en $(6, 6)$, reste 0)	Pic isolé devient top-hat ; vallée isolée devient black-hat ; les deux apparaissent dans le gradient

```
%%writefile EP04_08.py
# Code 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.fr/cap04/sim-ep0408-gradiente.html

Figure 4.37: Simulateur EP04_08: Gradient morphologique, Top-hat et Black-hat

4.9.9 EP04_09 Transformée de distance et le « cœur » de l'objet

En **robotique mobile**, lors de la planification d'un itinéraire dans un couloir, le robot souhaite savoir non seulement *où* se trouve l'espace libre, mais aussi **à quelle distance** chaque point libre se trouve du mur le plus proche. Les chemins les plus sûrs tendent à passer par le « cœur » du couloir, loin des obstacles.

La **transformée de distance morphologique** attribue à chaque pixel une valeur représentant sa distance jusqu'au bord le plus proche, selon la métrique définie par l'élément structurant. Les pixels proches du bord reçoivent des valeurs faibles, tandis que les pixels plus internes reçoivent des valeurs plus élevées. Le pixel de valeur maximale correspond à la région la plus protégée de l'objet, souvent associée à son centre morphologique.

Voir dans Figure 4.38 une simulation de cet EP.

4.9.9.1 📋 Directives d'implémentation

1. **Dimensions de l'image :** lire les entiers L (lignes) et C (colonnes) de l'image f.
2. **Dimensions de B :** lire les entiers L_B (lignes) et C_B (colonnes) de l'élément structurant.
3. **Élément structurant :** lire la matrice b, contenant la valeur 0 au centre et des valeurs négatives aux autres positions.
4. **Image :** lire la matrice binaire f (valeurs 0 ou 1), ligne par ligne.
5. **Préparation :** multiplier l'image par $L \times C$, en garantissant que les pixels internes ont une valeur initiale suffisamment élevée pour la propagation des distances.
6. **Transformée de distance :** calculer la matrice des distances en utilisant la méthode `mm.dist1(f,b)`.
7. **Sortie :** afficher la matrice résultante de la transformée de distance.

4.9.9.2 📌 Contraintes de calcul

- Utiliser l'implémentation de l'érosion pondérée fournie par la bibliothèque.
- L'élément structurant peut contenir des valeurs négatives arbitraires.
- La transformée doit être obtenue par l'application itérative d'érosions pondérées jusqu'à atteindre un point fixe.

⚠️ **Note cruciale sur la lecture des matrices :** Comme l'élément structurant peut contenir des entiers négatifs (par exemple, `-1` et `-99`), **ne pas utiliser la fonction `mm.readImg` pour lire la matrice** b. Cette

fonction convertit les données en type `uint8`, provoquant un *underflow* et corrompant les valeurs négatives. Lire les L_B lignes de b manuellement en utilisant le type standard `int`. L'image f peut continuer à être lue normalement par `mm.readImg`.

4.9.9.3 🧠 Fondement théorique

Concept	Signification	Impact visuel
dist(y, x)	Distance morphologique jusqu'au bord le plus proche selon la métrique définie par b	Les pixels plus internes reçoivent des valeurs plus élevées
Valeur maximale	Pixel le plus éloigné du bord	Se rapproche du centre morphologique de l'objet
Élément structurant pondéré	Définit les coûts de déplacement entre pixels voisins	Détermine la métrique de distance utilisée
Objets fins	Régions étroites de l'objet	Produisent des valeurs de distance faibles

4.9.9.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : entier L.
- Ligne 2 : entier C.
- Ligne 3 : entier L_B.
- Ligne 4 : entier C_B.
- Les L_B lignes suivantes : éléments entiers de la matrice b.
- Les L lignes suivantes : éléments binaires (0 ou 1) de la matrice f.

⚠️ **Note d'implémentation :** Les éléments de la matrice f (0 ou 1) doivent être multipliés par **255** pour générer une image binaire appropriée (0 et 255) avant d'appliquer la Transformée de Distance (TD).

Sortie :

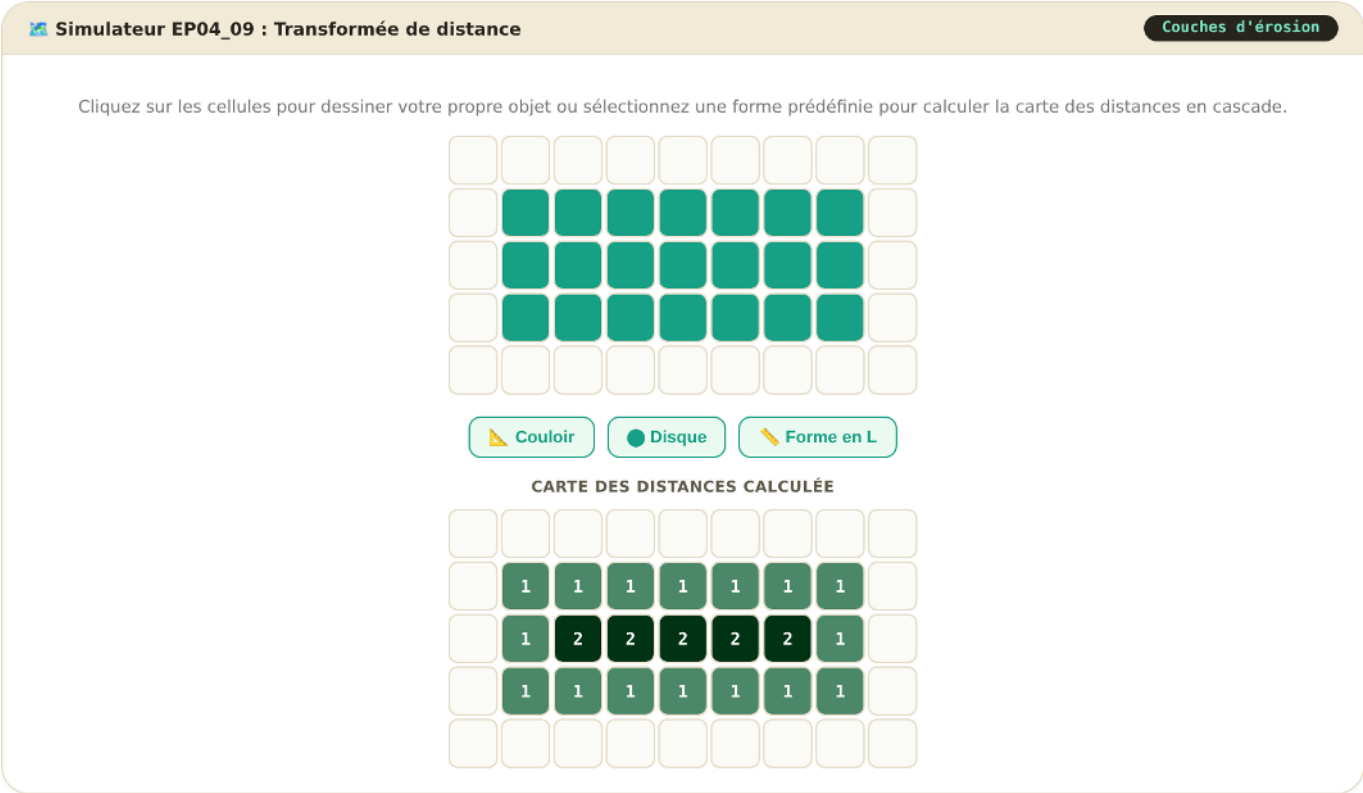
- Matrice de la transformée de distance en L lignes et C colonnes.

4.9.9.5 📌 Exemple

Entrée	Sortie	Observation
5 9 3 3 -99 -1 -99 -1 0 -1 -99 -1 -99 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 2 2 2 2 2 1 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0	Résultat de la transformée de distance.

Note : la valeur `-99` agit comme une approximation pratique de $-\infty$, empêchant la propagation par les

diagonales. Ainsi, seuls les voisins horizontaux et verticaux contribuent à la distance, produisant la distance de Manhattan.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-ep0409-distancia.html

Figure 4.38: Simulateur EP04_09: Transformée de Distance (Couches d'Érosion)

```
%%writefile EP04_09.py
# Code Python
```

```
Overwriting EP04_09.py
```

```
TestSuite("EP04_09.py").run()
```

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

4.9.10 EP04_10 🪙 Séparation des *blobs*, étiquetage et descripteurs

Dans une **ligne de production de pièces de monnaie**, il est courant que les pièces se touchent sur le tapis roulant, formant une seule tache connectée dans l'image — un comptage naïf donnerait un nombre erroné. La solution classique combine des opérations morphologiques et une analyse de connectivité : d'abord, une **érosion** réduit ou rompt les connexions fragiles entre les objets, puis l'**étiquetage des composantes connexes** sépare chaque objet en une région distincte. Enfin, des **descripteurs géométriques** (aire et boîte englobante) résument chaque composante trouvée.

Voir Figure 4.39 pour une simulation de cet EP.

4.9.10.1 📋 Directives d'implémentation

1. **Dimensions de l'image :** lire les entiers L (lignes) et C (colonnes) de f.
2. **Dimensions de B :** lire les entiers L_B (lignes) et C_B (colonnes) de l'élément structurant.
3. **Élément structurant :** lire la matrice B, contenant des valeurs 0 ou 1, ligne par ligne.

4. **Données :** lire la matrice binaire f (valeurs 0 ou 1), ligne par ligne.
5. **Séparation :** calculer
$$f_{ero} = f \ominus B$$
en utilisant une érosion binaire plane (comme dans l'EP04_04), en éliminant les connexions fragiles entre les objets.
6. **Étiquetage :** sur f_{ero}, identifier les composantes connexes en utilisant la connectivité définie par le voisinage B. L'étiquetage doit suivre un balayage *raster* : lorsqu'un pixel 1 non encore étiqueté est trouvé, attribuer un nouveau label entier croissant à partir de 1 et propager ce label à toute la région connexe.
7. **Descripteurs :** pour chaque label k, calculer :
   - **Aire :** nombre de pixels appartenant au label ;
   - **Boîte englobante :**
$$(y_{min}, x_{min}, y_{max}, x_{max})$$
8. **Sortie :** afficher le nombre total de labels, puis une ligne par label au format :
$$k,\ \text{aire},\ y_{min},\ x_{min},\ y_{max},\ x_{max}$$

4.9.10.2 📌 Contraintes computationnelles

- L'érosion doit être appliquée avant l'étiquetage.
- La connectivité est fixe et définie par le voisinage ci-dessus.
- L'élément structurant B n'interfère pas avec la connectivité de l'étiquetage.
- Aucun padding à aucune étape.
- L'ordre des labels suit la première découverte en balayage *raster*.

4.9.10.3 🧠 Fondement théorique

Concept	Signification	Impact
Pont fin	Connexion étroite entre objets	Peut être supprimé par l'érosion morphologique
Connectivité	Définie par l'ensemble $\mathcal{N}(y, x)$	Détermine quels pixels appartiennent à la même composante
Aire	Nombre de pixels par composante	Estimation directe de la taille de l'objet
Boîte englobante	Extension spatiale du label	Résumé géométrique de la composante

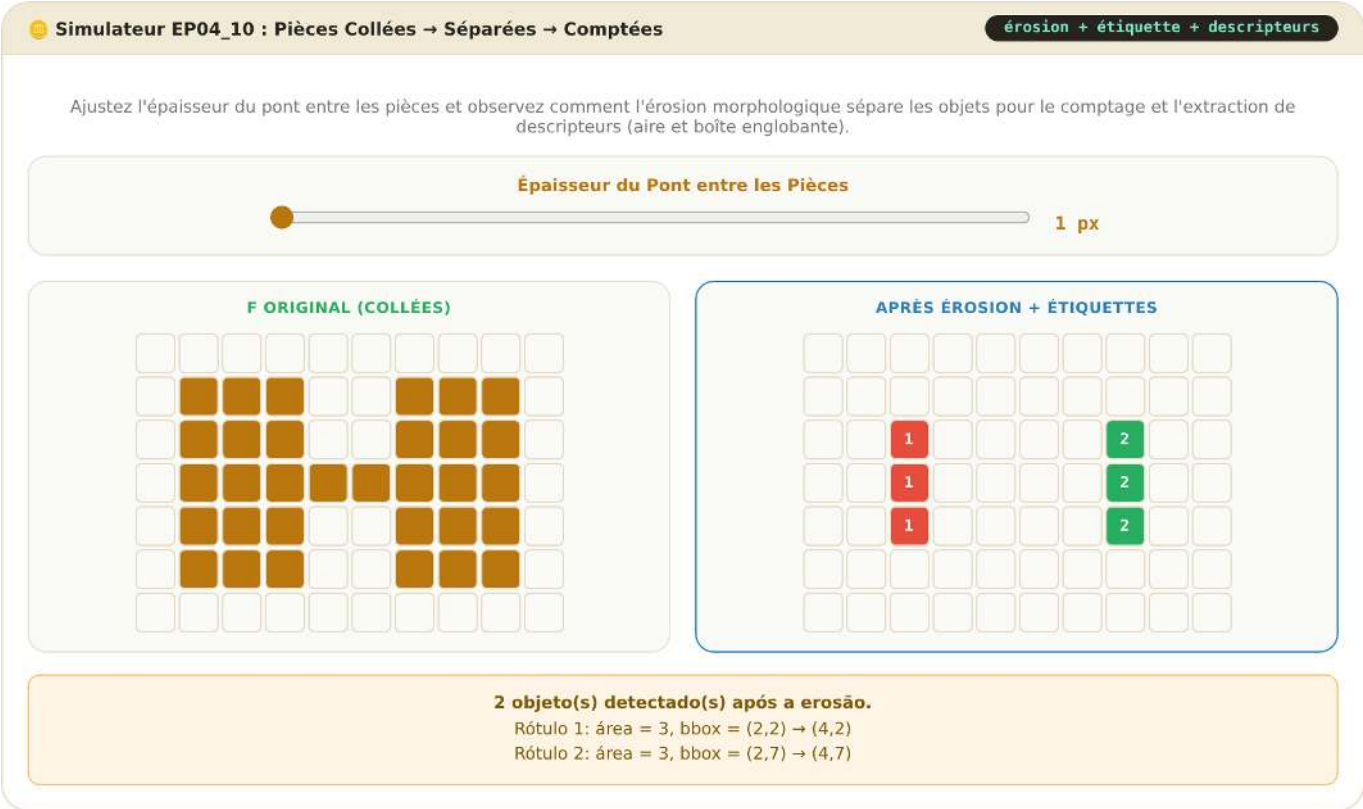
4.9.10.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : entier L
- Ligne 2 : entier C
- Ligne 3 : entier L_B
- Ligne 4 : entier C_B
- Les L_B lignes suivantes : matrice B
- Les L lignes suivantes : matrice f

Sortie :

- Ligne 1 : nombre total de labels trouvés
- Lignes suivantes :
$$k,\ \text{aire},\ y_{min},\ x_{min},\ y_{max},\ x_{max}$$



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap04/sim-ep0410-rotulacao.html

Figure 4.39: Simulateur EP04_10: Séparation des Blobs, Étiquetage et Descripteurs

```
%%writefile EP04_10.py
# Code Python
```

```
Overwriting EP04_10.py
```

```
TestSuite("EP04_10.py").run()
```

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

Chapitre 5

Capítulo 5 — Segmentation d'images

Exécuter Colab Ouvrir GitHub

5.1 5.1 Introduction

La segmentation d'images est l'une des étapes les plus importantes dans le traitement numérique des images. Elle consiste à partitionner une image en régions homogènes, facilitant ainsi l'analyse et l'interprétation du contenu. Ce chapitre présente les principales techniques de segmentation, en mettant l'accent sur les méthodes basées sur les seuils, la détection de contours et la croissance de régions.

5.2 5.2 Segmentation par seuillage

Le seuillage est une méthode simple et efficace pour segmenter une image lorsque les objets d'intérêt présentent une intensité de gris distincte par rapport au fond. Le principe consiste à sélectionner un seuil T et à classer chaque pixel en fonction de sa valeur d'intensité :

- Si $f(x, y) \geq T$, le pixel est considéré comme appartenant à l'objet.
- Sinon, le pixel est attribué au fond.

Le choix du seuil T peut être effectué manuellement ou à l'aide de méthodes automatisées, comme la méthode d'Otsu, qui maximise la variance interclasse.

5.3 5.3 Détection de contours

La détection de contours vise à identifier les frontières entre différentes régions. Elle repose sur l'analyse des variations d'intensité, souvent calculées à l'aide d'opérateurs de dérivée, tels que les filtres de Sobel, Prewitt ou Canny. Le gradient d'une image $G(x, y)$ peut être exprimé par :

$$G(x, y) = \sqrt{G_x^2 + G_y^2}$$

où G_x et G_y sont les dérivées partielles dans les directions horizontale et verticale.

5.4 5.4 Croissance de régions

La croissance de régions est une technique qui regroupe des pixels voisins présentant des caractéristiques similaires (intensité, texture, couleur). Elle débute avec des germes (pixels initiaux) et s'étend en ajoutant les pixels voisins qui respectent un critère de similarité, comme la différence d'intensité inférieure à un seuil prédéfini.

5.5 5.5 Exemples pratiques

Dans cette section, nous appliquons les techniques présentées à des images réelles. Les exemples sont implémentés en Python à l'aide des bibliothèques OpenCV et scikit-image. Chaque exemple est accompagné d'explications sur les paramètres et les résultats obtenus.

5.6 5.6 Résumé

La segmentation constitue un fondement essentiel pour de nombreuses applications de vision par ordinateur, telles que la reconnaissance d'objets, l'analyse médicale et la navigation autonome. Le choix de la méthode dépend du type d'image et des objectifs de l'application.

Chapitre 6

Transformées et Compression

Dans les chapitres précédents, toutes les opérations ont été réalisées **dans le domaine spatial**, où les algorithmes agissent directement sur les valeurs d'intensité des pixels.

Ce chapitre présente une approche complémentaire : le **domaine fréquentiel**, dans lequel l'image est représentée par les variations spatiales d'intensité, et non seulement par les valeurs individuelles des pixels.

Le concept de **fréquence spatiale** décrit la rapidité avec laquelle l'intensité varie le long de l'image. Les variations lentes correspondent aux **basses fréquences**, tandis que les contours, les détails fins et les bruits correspondent aux **hautes fréquences**.

Cette représentation repose sur le fait que toute image numérique discrète peut être décomposée en une combinaison de **fonctions orthogonales**. La **Transformée de Fourier** utilise une **base d'exponentielles complexes bidimensionnelles** (équivalentes à des sinusoïdes avec une orientation et une fréquence spécifiques). D'autres transformées, comme la **Transformée en Cosinus (DCT)** et la **Transformée *Wavelet* (DWT)**, utilisent différentes familles de fonctions de base — des cosinus bidimensionnels dans le cas de la DCT, et des fonctions à support compact dans le cas des *wavelets*.

Parmi les principales applications de cette représentation, on distingue :

1. **Le filtrage dans le domaine fréquentiel**, pour atténuer ou rehausser certaines bandes de fréquences ;
2. **L'analyse multirésolution au moyen de transformées *wavelet***, qui représente les structures à différentes échelles ;
3. **La compression d'images**, par la réduction du nombre de coefficients nécessaires pour représenter l'image.

6.1 Objectifs

À la fin de ce chapitre, vous serez capable de :

- **Interpréter le spectre de Fourier** d'une image, en distinguant l'amplitude, la phase et les composantes de fréquence ;
- **Appliquer le théorème de convolution** pour effectuer un filtrage dans le domaine fréquentiel à l'aide de la transformée de Fourier rapide (FFT) ;
- **Concevoir et analyser des filtres dans le domaine fréquentiel**, en comprenant le fonctionnement des filtres passe-bas, passe-haut et *coupe-bande* ;
- **Comprendre l'analyse multirésolution par transformées en *ondelettes*** et son application à la représentation hiérarchique des images ;
- **Décrire le processus de compression d'images**, y compris la transformée en cosinus discrète (DCT) et la quantification des coefficients ;
- **Choisir des formats de stockage d'images**, tels que JPEG, PNG et WebP, en fonction des exigences de l'application.

6.2 Configuration de l'environnement

```
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
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0
```

6.3 Transformada de Fourier Discreta 2D

L'analyse de Fourier repose sur le principe selon lequel tout signal périodique peut être représenté comme une somme de fonctions sinusoïdales de différentes fréquences, amplitudes et phases. Ce concept s'applique également aux images numériques, permettant de les représenter dans le **domaine fréquentiel** plutôt que dans le domaine spatial.

La Figure 6.1 illustre cette décomposition pour un signal unidimensionnel. Dans le cas d'une image, la Transformée de Fourier Discrète (TFD) convertit la matrice d'intensités $f(x, y)$ en un ensemble de coefficients décrivant la contribution des différentes fréquences spatiales présentes dans l'image.

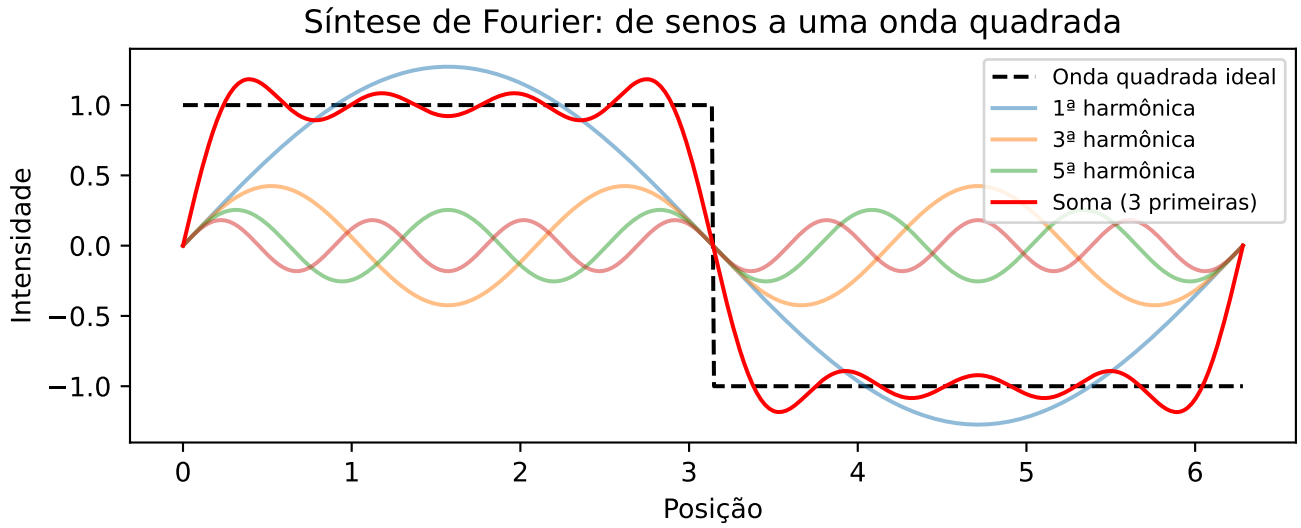


Figure 6.1: Décomposition de Fourier 1D : une onde carrée (ligne pointillée) est approximée par la somme des premières sinusoïdes (lignes colorées). Plus il y a de termes, meilleure est l'approximation.

6.3.1 Simulateur : Reconstruire des signaux avec des sinusoïdes

Avant d'étudier les images bidimensionnelles, le simulateur de la Figure 6.2 illustre le principe de l'analyse de Fourier pour des signaux unidimensionnels : **une forme d'onde peut être approchée par la somme de sinusoïdes de différentes fréquences et amplitudes**.

À mesure que de nouveaux termes sont ajoutés, la somme des sinusoïdes (courbe noire) se rapproche de la forme d'onde de référence (en pointillés). Le graphique inférieur présente le spectre d'amplitudes, indiquant la contribution de chaque fréquence à la reconstruction du signal.

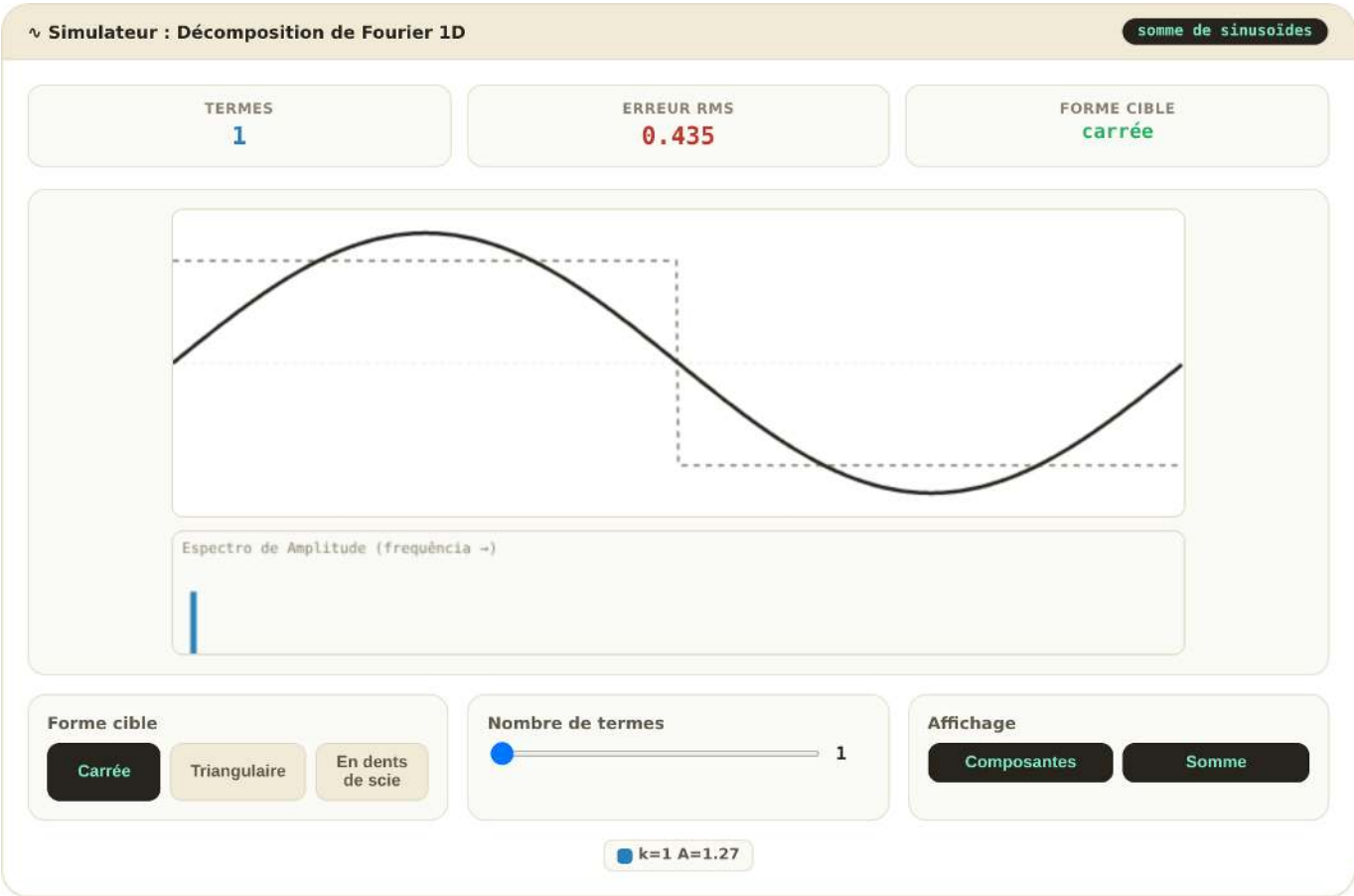
Activité

Explorez le simulateur et répondez :

1. Combien de termes sont nécessaires pour obtenir une bonne approximation de l'onde carrée ?
2. Laquelle des trois formes d'onde converge le plus rapidement ? Justifiez votre réponse.
3. Comment le spectre d'amplitudes se modifie-t-il en remplaçant l'onde carrée par l'onde triangulaire ?

Réponses

1. Combien de termes sont nécessaires pour une bonne approximation de l'onde carrée ?
Avec environ 15 à 20 termes, la forme de l'onde se rapproche déjà bien de la référence. Cependant, à proximité des discontinuités subsiste une petite oscillation, connue sous le nom de **phénomène de Gibbs**, qui ne disparaît pas même avec l'ajout de termes supplémentaires.
2. Quelle forme converge le plus rapidement ? Pourquoi ?
L'**onde triangulaire** converge plus rapidement, car les amplitudes de ses harmoniques décroissent plus vite que celles de l'onde carrée et de l'onde en dents de scie. Par conséquent, peu de termes suffisent déjà à produire une bonne approximation.
3. Comment le spectre change-t-il entre l'onde carrée et l'onde triangulaire ?
Toutes deux ne possèdent que des **harmoniques impairs**, mais, dans l'onde triangulaire, les amplitudes diminuent beaucoup plus rapidement. Ainsi, peu d'harmoniques suffisent pour reconstruire le signal avec une bonne précision.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap05/sim-05-freq.html

Figure 6.2: Simulateur interactif de la décomposition de Fourier 1D : visualisation de la somme de sinusoïdes avec différentes fréquences, amplitudes et phases. Ajoutez des termes et observez la convergence vers des formes d'onde arbitraires.

6.3.2 Interprétation du spectre de fréquences

En appliquant la Transformée de Fourier Discrète (TFD) à une image et en visualisant le module de ses coefficients (voir Figure 6.5), on obtient le **spectre de magnitude**, qui montre la distribution des fréquences spatiales présentes dans l'image.

Le coefficient situé à l'origine de la TFD, appelé **composante continue** (*Direct Current*), correspond à la fréquence nulle et représente l'intensité moyenne de l'image. Par convention, ce coefficient est stocké dans le coin supérieur gauche du spectre. Pour faciliter son interprétation, on applique l'opération **FFT Shift**, qui déplace la composante continue vers le centre de l'image. Après ce décalage, les basses fréquences se concentrent dans la région centrale, tandis que les hautes fréquences se trouvent près des bords, comme le résume la Table 6.1.

Tableau 6.1: Correspondance entre les régions du spectre de magnitude après application du FFT Shift.

Région du spectre	Composantes prédominantes	Exemples dans l'image
Centre (basses fréquences)	Variations spatiales lentes	Éclairage, régions homogènes et formes globales
Région intermédiaire (fréquences moyennes)	Variations à échelle intermédiaire	Textures et motifs répétitifs
Bords (hautes fréquences)	Variations spatiales rapides	Contours, détails fins et bruit

Cette organisation facilite l'interprétation du spectre et la conception de filtres. L'atténuation des basses fréquences réduit les variations globales d'intensité, tandis que l'atténuation des hautes fréquences lisse l'image en réduisant les détails fins et une partie du bruit.

6.3.3 L'Expérience de la Grille : Construire une Image à partir d'un Unique Coefficient

Avant de présenter la formulation mathématique de la Transformée de Fourier Discrète (TFD), il est utile d'analyser son inverse, appelée Transformée de Fourier Discrète Inverse (TFDI). Considérons un spectre où tous les coefficients sont nuls, à l'exception d'un seul. Un exemple de cette construction est présenté dans le code de la Figure 6.3 et peut être exploré de manière interactive dans le simulateur de la Figure 6.4..

L'image reconstruite est une sinusoïde bidimensionnelle. La position du coefficient dans le spectre détermine son **orientation** et sa **fréquence spatiale**, tandis que sa magnitude et sa phase définissent, respectivement, son amplitude et son déplacement spatial. Ainsi, chaque coefficient de la TFD représente une composante sinusoïdale, et l'image originale peut être reconstruite par la somme de toutes ces composantes.

```
import cv2

N_grid = 100
espectro_vazio = np.zeros((N_grid, N_grid), dtype=complex)

# Allumant un seul point (fréquence) hors du centre
u0, v0 = 10, 5
espectro_vazio[N_grid//2 - v0, N_grid//2 - u0] = 1000

# Retour au domaine spatial (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)

# Mise en évidence visuelle du point
espectro_color = cv2.cvtColor(espectro_vis, cv2.COLOR_GRAY2BGR)
cv2.circle(espectro_color, (N_grid//2 - u0, N_grid//2 - v0), 2, (0, 0, 255), -1)
```

```
mm.show([espectro_color, onda_vis],
        titles=["Spectre (1 point actif)", "Onde 2D Résultante (IDFT)"],
        cols=2, figsize=(10, 4))
```

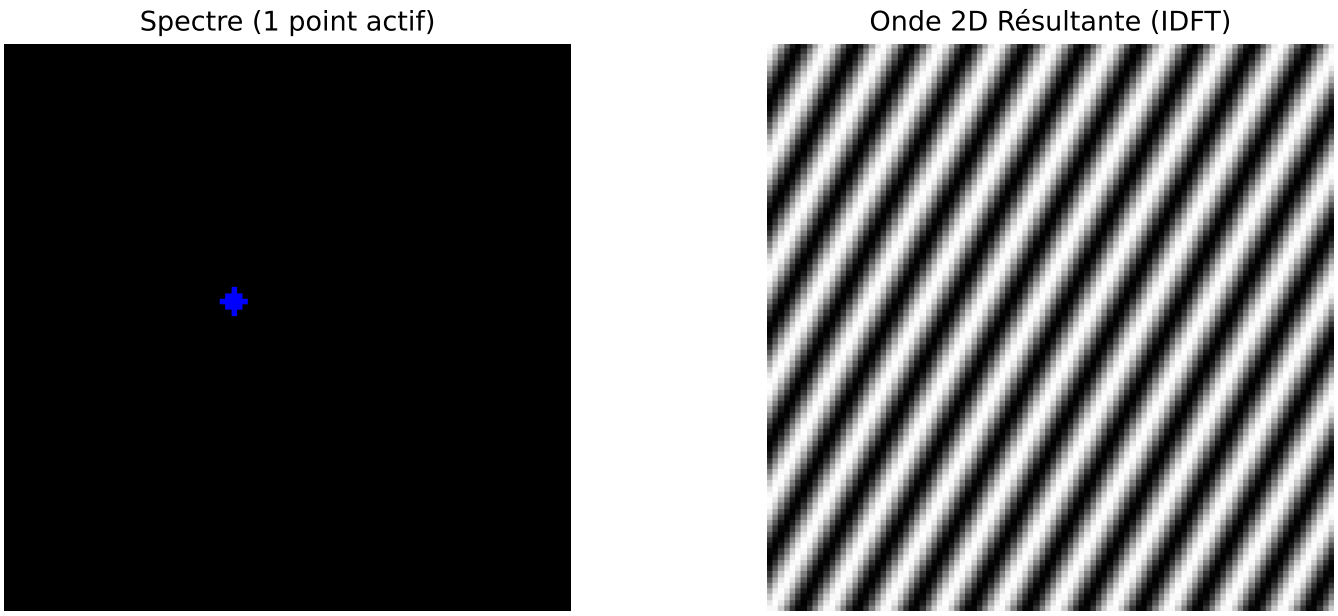
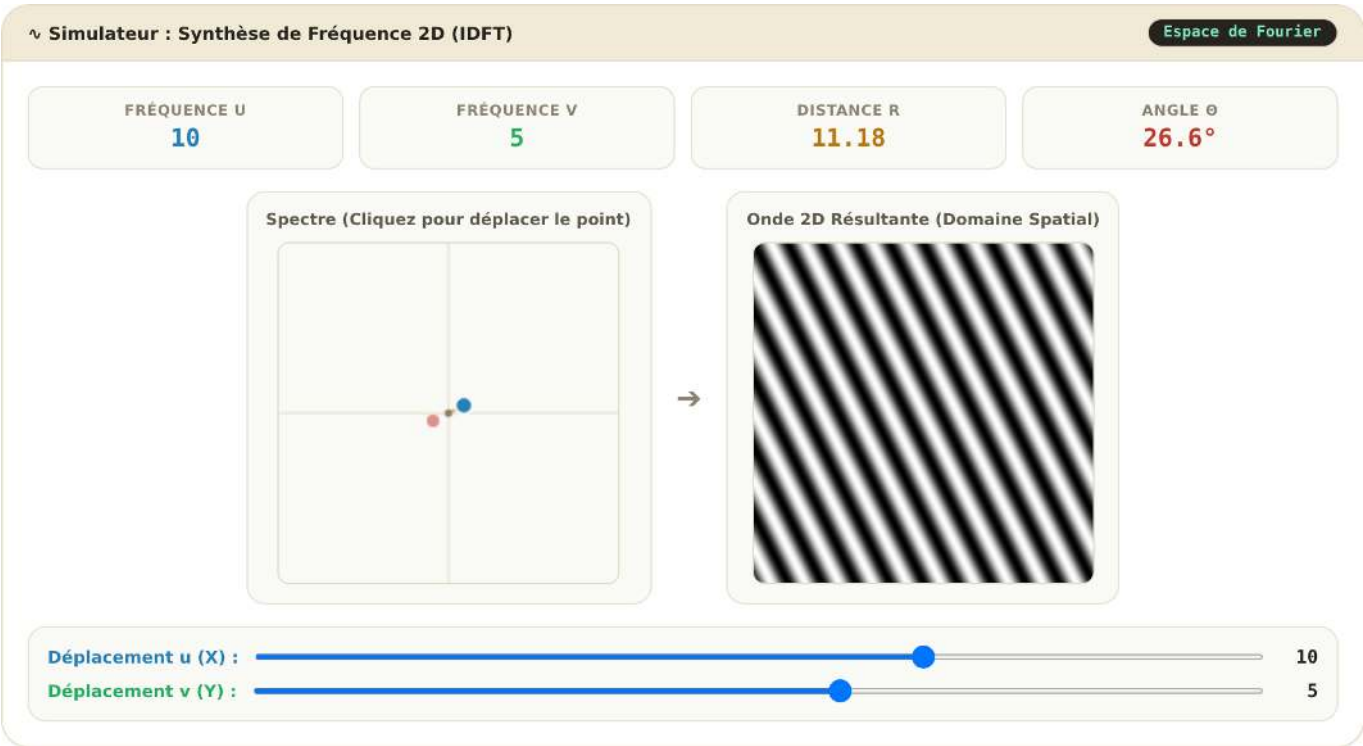


Figure 6.3: Toute fréquence dans le spectre (point isolé) correspond à une onde sinusoïdale 2D tournée dans le domaine spatial.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap05/sim-05-grade-2d.html

Figure 6.4: Simulateur interactif de la synthèse de Fourier 2D. Modifiez la position horizontale (u) et verticale (v) du coefficient dans le spectre de fréquences centré et observez comment la distance par rapport au centre dicte la fréquence spatiale (épaisseur) et l'angle dicte l'orientation de l'onde sinusoïdale générée.

6.3.4 Définition mathématique

Considérons une image $f(x, y)$ de dimensions $M \times N$. Sa **Transformée de Fourier discrète 2D** (TFD) est définie par :

$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y)\, e^{-j2\pi(\frac{ux}{M} + \frac{vy}{N})} \tag{6.1}$$

où $u = 0, 1, \ldots, M-1$ et $v = 0, 1, \ldots, N-1$ représentent les fréquences discrètes dans les directions horizontale et verticale, respectivement. Le terme exponentiel correspond à une sinusoïde bidimensionnelle, dont la fréquence et l'orientation sont déterminées par les indices (u, v).

La **Transformée de Fourier discrète inverse 2D** (TFDI) reconstruit l'image originale à partir de ses coefficients :

$$f(x, y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v)\, e^{j2\pi(\frac{ux}{M} + \frac{vy}{N})} \tag{6.2}$$

Les équations Équation 6.1 et Équation 6.2 montrent que la TFD et la TFDI forment une paire de transformations : la première convertit l'image dans le domaine fréquentiel, tandis que la seconde reconstruit exactement l'image originale à partir de ses coefficients.

À propos du symbole j

Le terme j désigne l'**unité imaginaire**, définie par $j^2 = -1$. En ingénierie et en traitement du signal, on adopte j plutôt que i afin d'éviter toute confusion avec la notation du courant électrique. Son utilisation dans l'exponentielle complexe, régie par la formule d'Euler ($e^{j\theta} = \cos\theta + j \sin\theta$), permet de représenter de manière compacte l'amplitude et la phase de chaque fréquence spatiale présente dans l'image.

Qu'est-ce que la composante DC ?

Le coefficient $F(0, 0)$, appelé **composante DC** (*courant continu*), est égal à la somme des intensités de tous les pixels de l'image (voir Figure 6.5) :

$$F(0, 0) = MN\, \bar{f},$$

où $\bar{f}$ est l'intensité moyenne de l'image. Ainsi, la composante DC représente le niveau moyen d'intensité et, dans la plupart des images naturelles, possède la plus grande amplitude du spectre.
Les autres coefficients représentent les variations autour de cette moyenne. Après application du **décalage de FFT**, la composante DC est déplacée vers le centre du spectre, concentrant les basses fréquences dans la région centrale et les hautes fréquences sur les bords.

6.3.5 Magnitude et Phase

Chaque coefficient de la Transformée de Fourier Discrète (TFD) est un nombre complexe et peut s'écrire sous la forme

$$F(u, v) = R(u, v) + j\, I(u, v),$$

où $R(u, v)$ et $I(u, v)$ correspondent respectivement aux parties **réelle** et **imaginaire** du coefficient. À partir de l'équation Équation 6.1, on obtient

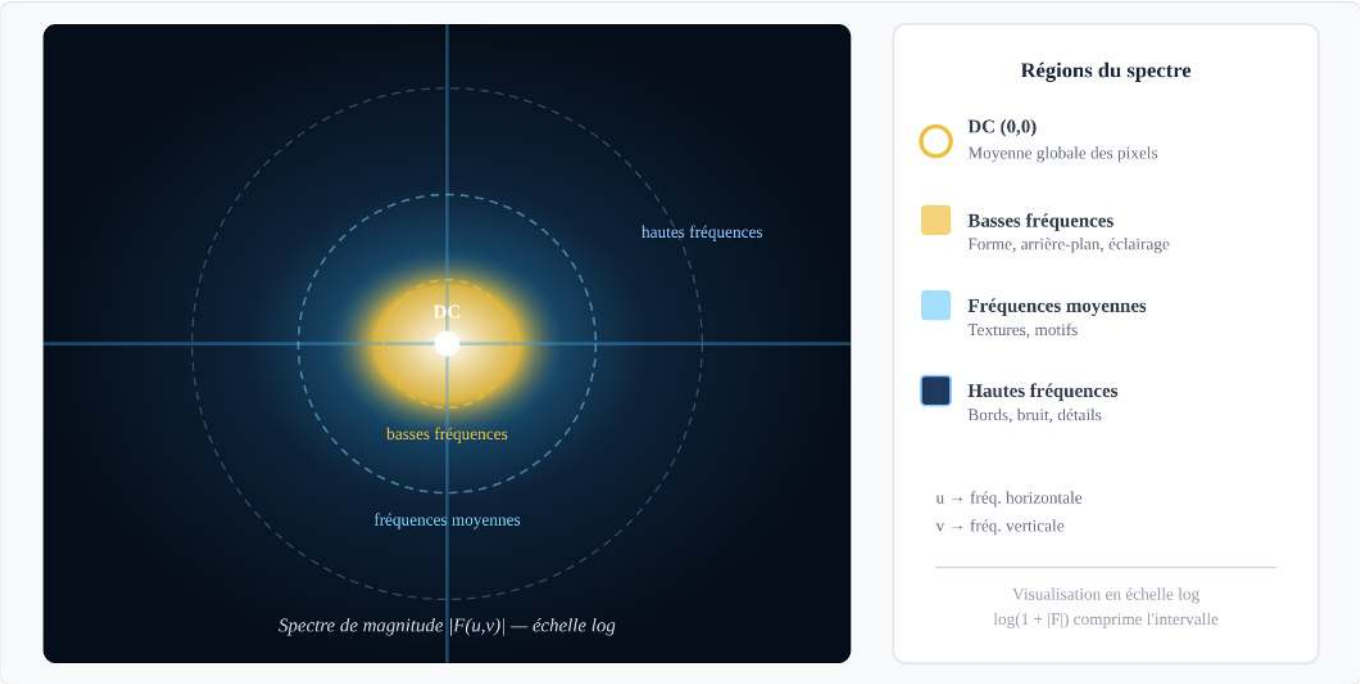


Figure 6.5: Diagramme conceptuel du spectre de Fourier 2D centré.

$$R(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \cos\Big(2\pi \left(\frac{ux}{M} + \frac{vy}{N}\right)\Big),$$

et

$$I(u,v) = -\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \sin\Big(2\pi \left(\frac{ux}{M} + \frac{vy}{N}\right)\Big).$$

À partir de cette représentation, on définit deux grandeurs fondamentales :

- **Magnitude**, qui indique l'intensité de la composante fréquentielle,

$$|F(u,v)| = \sqrt{R(u,v)^2 + I(u,v)^2};$$

- **Phase**, qui détermine l'alignement (ou le décalage) spatial de la composante,

$$\phi(u,v) = \operatorname{atan2}(I(u,v),\, R(u,v)).$$

Ainsi, chaque coefficient peut également être écrit sous sa forme polaire,

$$F(u,v) = |F(u,v)|\, e^{j\phi(u,v)}.$$

Le spectre de Fourier peut donc être visualisé au moyen de deux images distinctes : le **spectre de magnitude**, généralement utilisé pour analyser la distribution des fréquences, et le **spectre de phase**, qui décrit l'organisation spatiale des composantes sinusoïdales.

Bien que le spectre de magnitude soit le plus utilisé pour l'inspection visuelle, la phase contient une grande partie des informations structurelles de l'image. La combinaison de la magnitude et de la phase permet de reconstruire exactement l'image originale au moyen de la TFD inverse.

6.3.6 O que a Magnitude e a Fase carregam?

6.3.7 Ce que portent la magnitude et la phase ?

Une démonstration classique consiste à combiner la magnitude d'une image avec la phase d'une autre et à reconstruire le résultat. Cette expérience met en évidence que :

- **La phase** préserve la structure spatiale de l'image, y compris la position des objets, leurs contours et leur géométrie. De petites modifications de la phase peuvent provoquer de grands changements visuels.
- **La magnitude** contrôle la manière dont l'énergie est distribuée entre les fréquences spatiales, influençant principalement le contraste et la texture.

Lorsqu'une image est reconstruite avec la magnitude de A et la phase de B, le résultat tend à **ressembler davantage à B qu'à A**, ce qui montre que la phase est le principal composant responsable de l'organisation spatiale de la scène. Cependant, la magnitude reste importante, car elle module le contraste des structures reconstruites. Ainsi, une reconstruction fidèle dépend de la combinaison cohérente entre magnitude et phase.

Un exemple de ce comportement est présenté dans la Figure 6.6.

ℹ Analogie avec l'audio : limites et précautions

La phase d'un signal joue des rôles distincts dans l'audio et les images :

- **Audio stéréo ou multicanal :** la phase relative entre les canaux est fondamentale pour la perception de la position des sources sonores, via les différences interaurales de temps (ITD, *Interaural Time Differences*).
- **Audio monaural :** la phase absolue exerce peu d'influence perceptuelle directe.
- **Images (TFD) :** la phase est le principal facteur responsable de l'organisation spatiale de la scène, tandis que la magnitude module le contraste et la distribution de l'énergie entre les fréquences.

Dans les deux domaines, la **magnitude** est liée à l'intensité des composantes de fréquence : en audio, elle influence le timbre et l'intensité perçue ; en images, elle influence le contraste et la texture.

```
#   Expérience : L'importance de la phase
#   Chargement de l'image
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))

# Créer une image B synthétique (motif géométrique)
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)

# Échange de phase
rec_A_mag_B_fase = np.real(np.fft.ifft2(np.abs(FA) * np.exp(1j * np.angle(FB))))
rec_B_mag_A_fase = np.real(np.fft.ifft2(np.abs(FB) * np.exp(1j * np.angle(FA))))
```

```
mm.show(
    [img_a, img_b, rec_A_mag_B_fase, rec_B_mag_A_fase],
    titles=["Image A", "Image B", "Mag(A) + Phase(B)", "Mag(B) + Phase(A)"],
    cols=4, figsize=(16, 4)
)

print("  La phase préserve les bords et les contours ; la magnitude contrôle le contraste et")
print("la texture. En audio stéréo, la phase affecte la localisation spatiale ; en")
print("images, elle détermine l'organisation de la scène.")
```

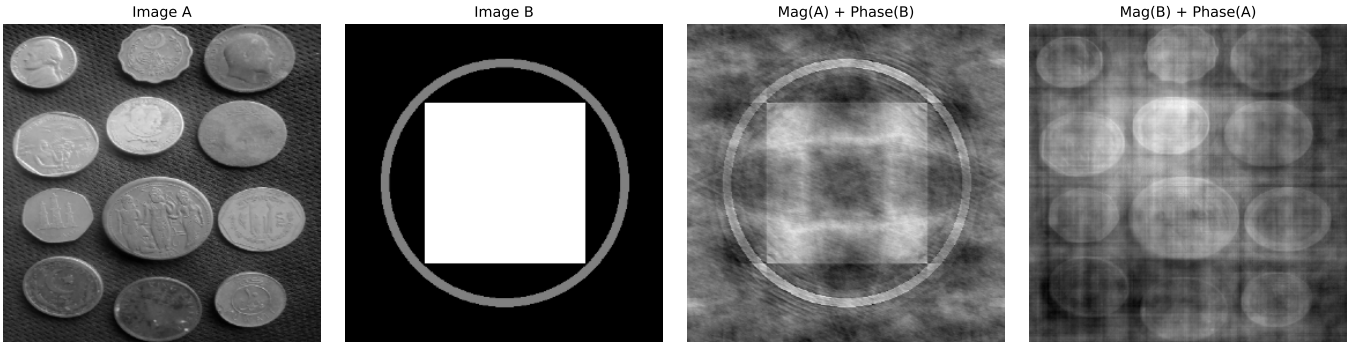


Figure 6.6: Expérience de changement de phase : Image A (pièces) et Image B (motif géométrique) reconstruites avec magnitudes et phases échangées. Le résultat montre que la structure visuelle est **beaucoup plus sensible à la phase** qu'à la magnitude : lorsque la phase de B est conservée, l'image résultante préserve l'organisation spatiale de B, même avec la magnitude de A. La magnitude, quant à elle, influence principalement le contraste et la texture. Observez que la qualité de la reconstruction n'est pas parfaite — des artefacts visibles sont présents —, soulignant l'interdépendance entre phase et magnitude pour une représentation fidèle de l'image.

```
  La phase préserve les bords et les contours ; la magnitude contrôle le contraste et
la texture. En audio stéréo, la phase affecte la localisation spatiale ; en
images, elle détermine l'organisation de la scène.
```

6.4 Théorème de convolution et stratégies de filtrage

Le **théorème de convolution** établit une relation fondamentale entre les domaines spatial et fréquentiel :

$$f(x,y) \circledast h(x,y) \overset{\mathcal{F}}{\longleftrightarrow} F(u,v)\,H(u,v) \tag{6.3}$$

où $\circledast$ représente la **convolution circulaire discrète**. Ainsi, la convolution entre une image $f(x,y)$ et un filtre $h(x,y)$ peut être remplacée par la multiplication de leurs spectres.

En pratique, pour obtenir le même résultat que la convolution linéaire effectuée dans le domaine spatial, on applique un **remplissage par zéros** (*zero-padding*) avant la Transformée de Fourier rapide (FFT), afin d'éviter les artefacts sur les bords de l'image.

Cependant, le filtrage dans le domaine fréquentiel n'est pas toujours l'alternative la plus efficace. Pour des filtres tels que le filtre gaussien et le filtre moyenneur (*Box Filter*), la propriété de **séparabilité** permet de réduire considérablement le coût computationnel de la convolution dans le domaine spatial.

6.4.1 *Noyau* séparable vs. non séparable

Un ***noyau* séparable** peut être écrit comme le produit externe de deux vecteurs unidimensionnels,

$$H = v\,h^T,$$

ce qui permet de remplacer la convolution bidimensionnelle par deux convolutions unidimensionnelles successives : l'une dans la direction horizontale et l'autre dans la direction verticale.

En revanche, un ***noyau* non séparable** n'admet pas cette décomposition et, par conséquent, sa convolution doit être réalisée directement sur le voisinage bidimensionnel.

En pratique, pour un *noyau* de dimension $K \times K$, la convolution directe exige K^2 multiplications par pixel, tandis qu'un *noyau* séparable ne requiert que $2K$ multiplications, réduisant ainsi considérablement le coût computationnel.

6.4.2 Analyse de l'efficacité computationnelle

Considérons une image de dimensions $M \times N$ et un filtre carré de taille $K \times K$. La Table 6.2 compare la complexité des principales stratégies de filtrage.

Tableau 6.2: Comparaison de la complexité de la convolution directe, séparable et via la Transformée de Fourier rapide (FFT).

Méthode de filtrage	Complexité asymptotique	Dépendance de K	Application typique
Spatiale non séparable	$\mathcal{O}(MNK^2)$	Quadratique	*Kernels* petits et non séparables
Spatiale séparable	$\mathcal{O}(MNK)$	Linéaire	Filtres gaussien et de moyenne
Via FFT	$\mathcal{O}(MN\log(MN))$	Indépendante de K	*Kernels* grands

Pour de petits *kernels*, la convolution spatiale, surtout lorsque le filtre est séparable, s'avère généralement plus efficace en raison du faible coût des opérations. À mesure que la taille du *kernel* augmente, le filtrage via FFT devient plus avantageux, car son coût ne dépend pratiquement pas de la dimension du filtre.

6.4.3 Discussion des résultats expérimentaux

Le graphique obtenu lors de l'essai avec l'image des pièces (2560×1920), présenté à la Figure 6.7, confirme le comportement prédit par l'analyse de complexité computationnelle.

1. **Convolution non séparable ($\mathcal{O}(MNK^2)$)**
   La convolution directe présente une croissance quadratique avec la taille du *kernel*. Pour de petites valeurs de K, le coût est faible, mais il augmente rapidement à mesure que le *kernel* grandit, devenant irréalisable pour des applications en temps réel.

2. **Filtrage par FFT ($\mathcal{O}(MN\log(MN))$)**
   Le coût de la FFT ne dépend que de la taille de l'image, étant indépendant de K. Par conséquent, ses performances restent approximativement constantes lorsque le *kernel* varie, ce qui la rend avantageuse pour les filtres de grande taille ou non séparables.

3. **Convolution séparable ($\mathcal{O}(MNK)$)**
   La décomposition du *kernel* en deux filtres unidimensionnels réduit significativement le coût computationnel. En pratique, cette approche tend à être la plus efficace pour les filtres séparables, en particulier dans les implémentations optimisées.

En général, le choix de la méthode dépend de la taille et de la structure du *kernel*. Les filtres séparables sont plus efficaces dans le domaine spatial, tandis que la FFT devient plus avantageuse pour les *kernels* de grande taille ou pour de multiples convolutions dans le domaine fréquentiel.

$$g = \mathcal{F}^{-1}[\mathcal{F}(f) \cdot \mathcal{F}(h)] \quad \text{(FFT)} \qquad g = f \circledast h \quad \text{(convolution directe)} \qquad g = (f \circledast v) \circledast h^T \quad \text{(séparable)} \tag{6.4}$$

où :

- $f(x, y)$ représente l'image d'entrée ;
- $h(x, y)$ est le *kernel* bidimensionnel du filtre ;
- v et h^T sont, respectivement, les vecteurs vertical et horizontal qui composent le *kernel* séparable.

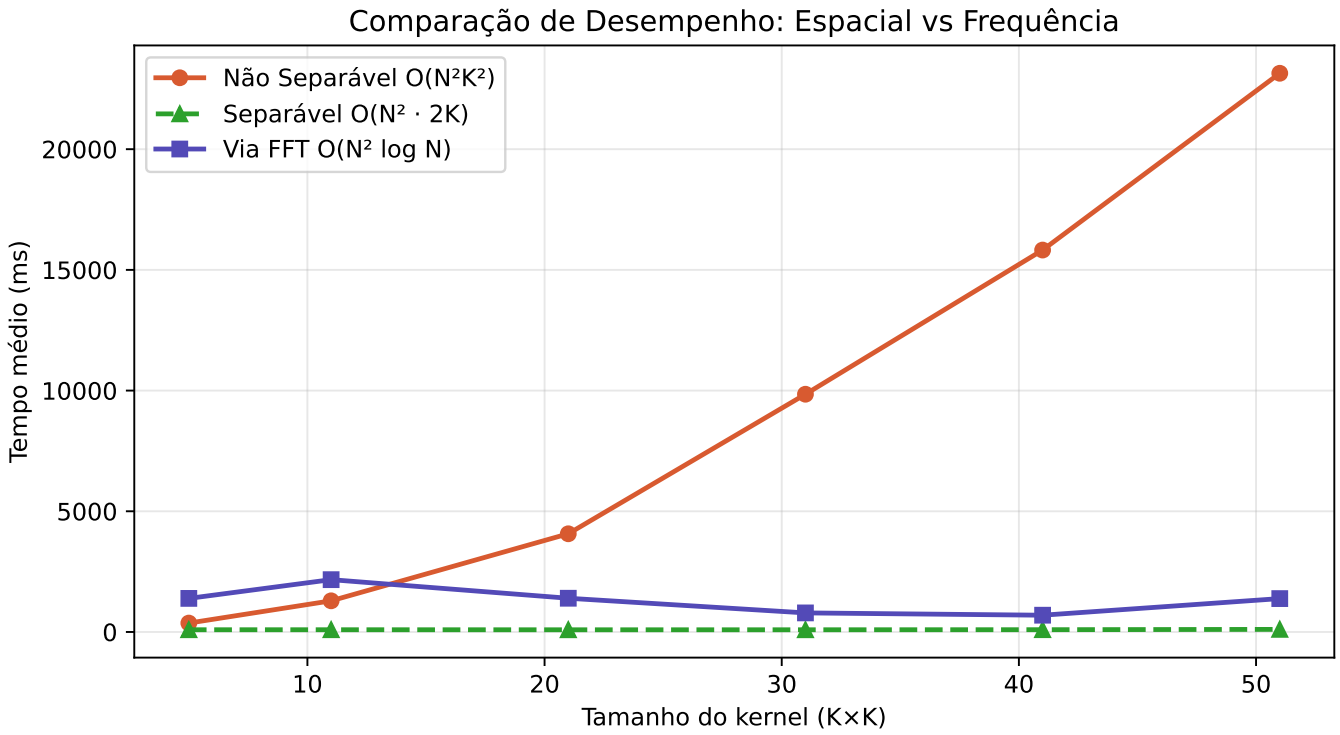


Figure 6.7: Comparaison d'efficacité : Convolution non séparable (spatiale 2D), séparable (spatiale 1D) et via FFT.

! Le problème de la convolution circulaire (*wrap-around*)

La Transformée de Fourier Discrète (TFD) suppose que l'image est **périodiquement étendue dans l'espace**, c'est-à-dire que ses bords se répètent indéfiniment.
Dans cette condition, la multiplication dans le domaine fréquentiel correspond à une **convolution circulaire** dans le domaine spatial. Par conséquent, des régions opposées de l'image (haut et bas, gauche et droite) interagissent artificiellement, comme illustré dans la Figure 6.8.
L'application d'un *zero-padding* avant la FFT réduit cet effet en étendant l'image avec des valeurs nulles sur les bords, rapprochant ainsi le résultat de la convolution linéaire. Ce comportement peut être interprété à la lumière du Théorème de Convolution, présenté dans la Figure 6.9.

```
#   Définir M et N
M, N = img_gray.shape  # ligne ajoutée

# Simulation d'un filtre de décalage brutal
H_shift = np.zeros_like(img_gray, dtype=complex)
for u in range(M):
    for v in range(N):
        H_shift[u, v] = np.exp(-1j * 2 * np.pi * (u*120/M + v*120/N))

# Filtrage SANS padding (provoque le 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", "Filtrage sans padding (Fuite)"], cols=2, figsize=(10, 4))
```

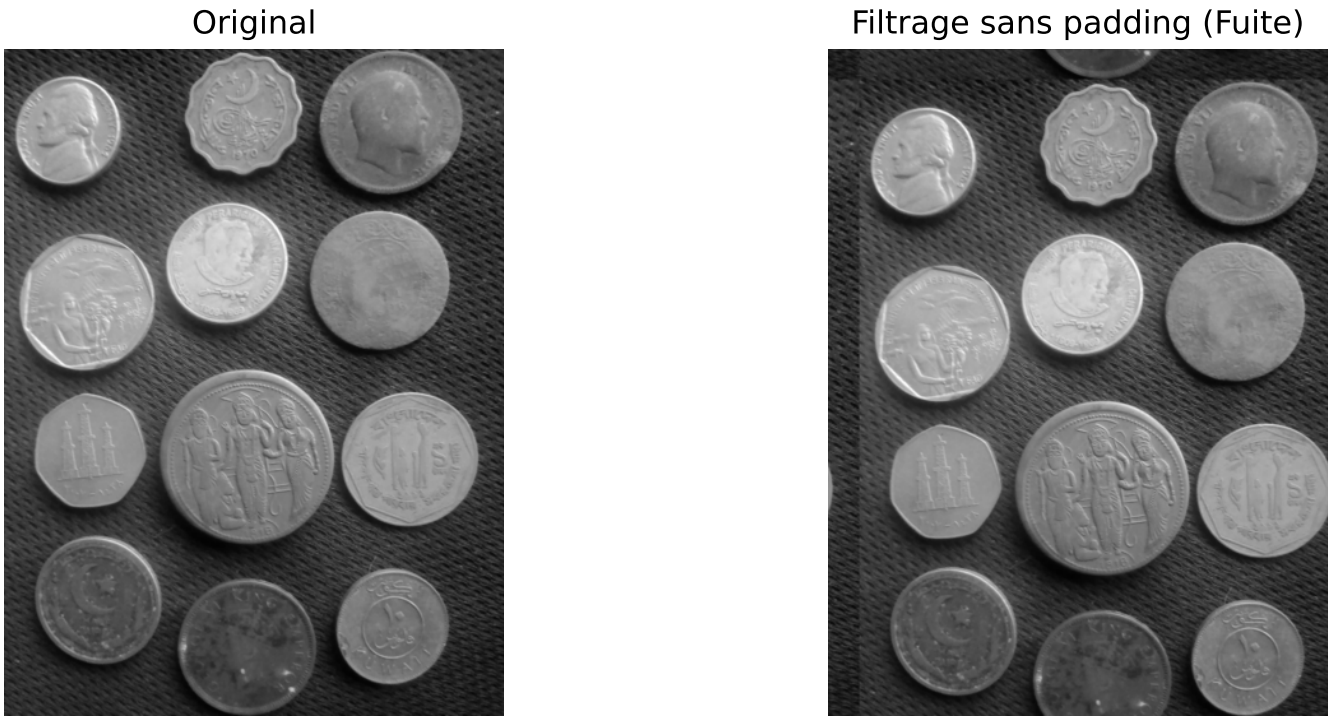


Figure 6.8: Sans *padding*, un décalage sévère fait fuir l'image vers le côté opposé (convolution circulaire).

```
#   Noyau gaussien 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)    # noyau 2D séparable

#   Méthode 1 : Convolution spatiale directe
f_float  = img_gray.astype(np.float64)
conv_esp = cv2.filter2D(f_float, -1, kernel, borderType=cv2.BORDER_CONSTANT)

#   Méthode 2 : Multiplication en fréquence (via FFT)
M, N     = f_float.shape
# Padding pour convolution linéaire (évite l'aliasing circulaire)
Mpad     = 2 ** int(np.ceil(np.log2(M + K - 1)))
Npad     = 2 ** int(np.ceil(np.log2(N + K - 1)))

# Positionne le noyau avec l'origine en (0,0) et padding avec des zéros
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))

# Recadrage pour compenser le décalage introduit par le positionnement du noyau
offset   = K // 2
```

```
conv_freq_crop = conv_freq[offset:offset+M, offset:offset+N]

#   Vérification numérique
diff = np.abs(conv_esp - conv_freq_crop)
print(f"Différence maximale  (|conv_esp - conv_freq|) : {diff.max():.2e}")
print(f"Différence moyenne   (|conv_esp - conv_freq|) : {diff.mean():.2e}")
print(f"→ Théorème de Convolution vérifié numériquement.")

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",
        "Convolution spatiale",
        "Multiplication en fréquence",
        f"Différence (max={diff.max():.1e})"
    ],
    cols=4, figsize=(16, 5)
)
```

```
Différence maximale  (|conv_esp - conv_freq|) : 2.56e-13
Différence moyenne   (|conv_esp - conv_freq|) : 2.76e-14
→ Théorème de Convolution vérifié numériquement.
```

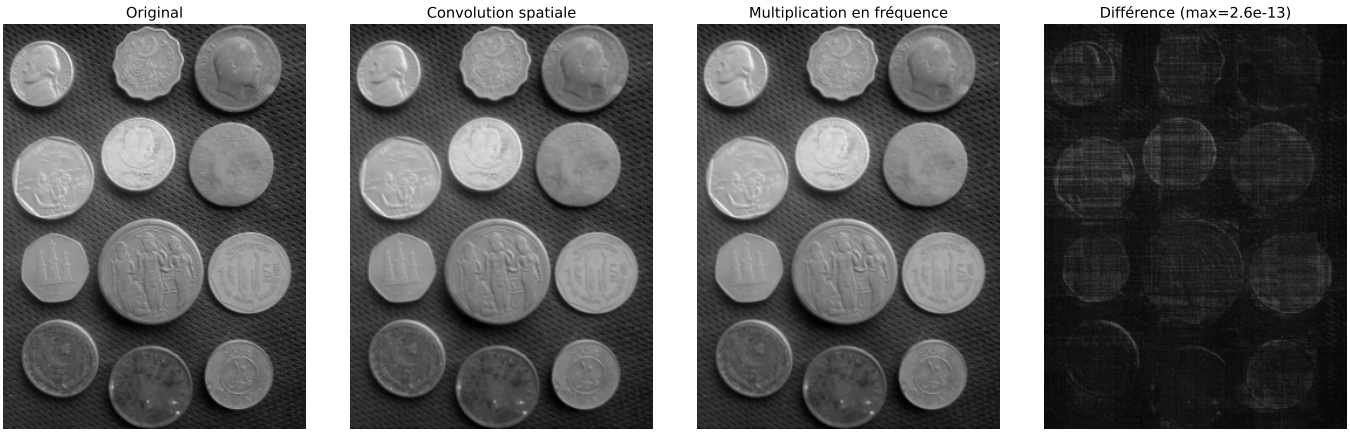


Figure 6.9: Vérification du Théorème de Convolution : la différence pixel à pixel entre la convolution spatiale (cv2.filter2D) et la multiplication en fréquence (FFT) est numériquement nulle — confirmant l'équivalence théorique.

À propos de la différence numérique

La différence résiduelle de l'ordre de 10^{-13} ne viole pas le Théorème de Convolution, mais reflète des **limitations computationnelles** inhérentes à l'arithmétique en virgule flottante (double précision, $\sim 10^{-16}$) et à l'**ordre des opérations** entre les deux méthodes :

- **Convolution spatiale :** somme pondérée des voisins avec des arrondis successifs.
- **Convolution fréquentielle :** implique trois transformées FFT et une multiplication complexe, sujette à des erreurs de troncature et de quantification.

Par conséquent, l'égalité théorique est exacte, mais l'implémentation numérique produit une différence pratiquement nulle (erreur relative $< 10^{-12}$), confirmant le théorème dans la précision de la machine.

6.5 Filtres dans le Domaine Fréquentiel

Un filtre dans le domaine fréquentiel peut être interprété comme une **fonction de transfert appliquée au spectre de l'image**. Dans cette représentation, chaque coefficient de fréquence est multiplié par une valeur comprise entre 0 et 1, qui détermine son atténuation ou sa préservation. La forme de cette fonction définit l'effet visuel du filtre.

Coupure brutale et *ringing*. Les filtres idéaux avec une transition instantanée à une fréquence de coupure D_0 produisent des discontinuités dans le domaine fréquentiel. Cette discontinuité se reflète dans le domaine spatial sous forme d'oscillations près des bords, connues sous le nom de *ringing*. Cet effet est associé à la convolution avec des fonctions à support infini dans l'espace, comme la fonction *sinc*, comme illustré dans la Figure 6.10.

Filtres à transition douce. Des alternatives telles que les filtres gaussien et de Butterworth lissent la transition entre les régions de passage et de rejet, réduisant ainsi le *ringing*. En contrepartie, ce lissage implique une frontière de séparation moins nette entre les fréquences préservées et atténuées.

```
# Simulation du Filtre Idéal en Fréquence (Cylindre) et de sa représentation Spatiale (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)

# Fréquence : Cylindre Idéal (1 au centre, 0 hors du rayon 20)
H_freq = np.zeros((N_grid, N_grid))
H_freq[D <= 20] = 1

# Espace : L'inverse résulte en la fameuse 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()
```

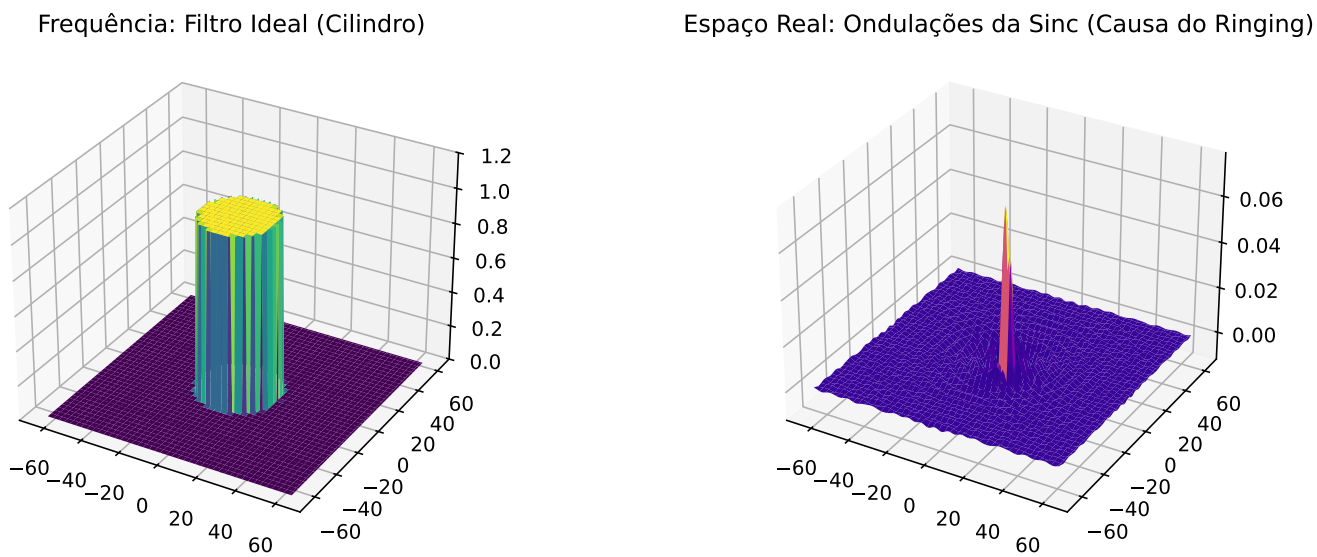


Figure 6.10: La Dualité Dangereuse : La coupure brutale en Fréquence (Cylindre) se transforme obligatoirement en Sinc spatiale. Ses ondulations provoquent le *ringing* fantôme sur les bords de l'image.

6.5.1 Filtres Passe-Bas

Les filtres passe-bas atténuent les composantes haute fréquence, ce qui entraîne un lissage de l'image et une réduction du bruit. Après la centralisation du spectre (FFT Shift), la distance de chaque point au centre est donnée par :

$$D(u,v) = \sqrt{\left(u - \frac{M}{2}\right)^2 + \left(v - \frac{N}{2}\right)^2} \tag{6.5}$$

Filtre idéal (LPFI) :

$$H_{\text{ideal}}(u,v) = \begin{cases} 1, & D(u,v) \leq D_0 \\ 0, & D(u,v) > D_0 \end{cases} \tag{6.6}$$

La coupure brutale à D_0 introduit des discontinuités dans le domaine fréquentiel, entraînant des oscillations dans le domaine spatial, connues sous le nom de *ringing*. Cet effet est associé à la convolution avec des fonctions à support infini.

Filtre gaussien (LPFG) :

$$H_{\text{gauss}}(u,v) = e^{-D^2(u,v)/(2\sigma^2)} \tag{6.7}$$

La régularité de la fonction gaussienne dans le domaine fréquentiel évite les discontinuités, ce qui élimine le *ringing* et produit une transition progressive entre les fréquences conservées et atténuées.

Filtre de Butterworth (LPFB) d'ordre n :

$$H_{\text{BW}}(u,v) = \frac{1}{1 + [D(u,v)/D_0]^{2n}} \tag{6.8}$$

Le paramètre n contrôle la régularité de la transition entre la transmission et la rejection des fréquences. Les petites valeurs produisent des transitions douces, tandis que les grandes valeurs rapprochent le comportement du filtre idéal, avec un risque accru de *ringing*. Un exemple comparatif est présenté à la Figure 6.11..

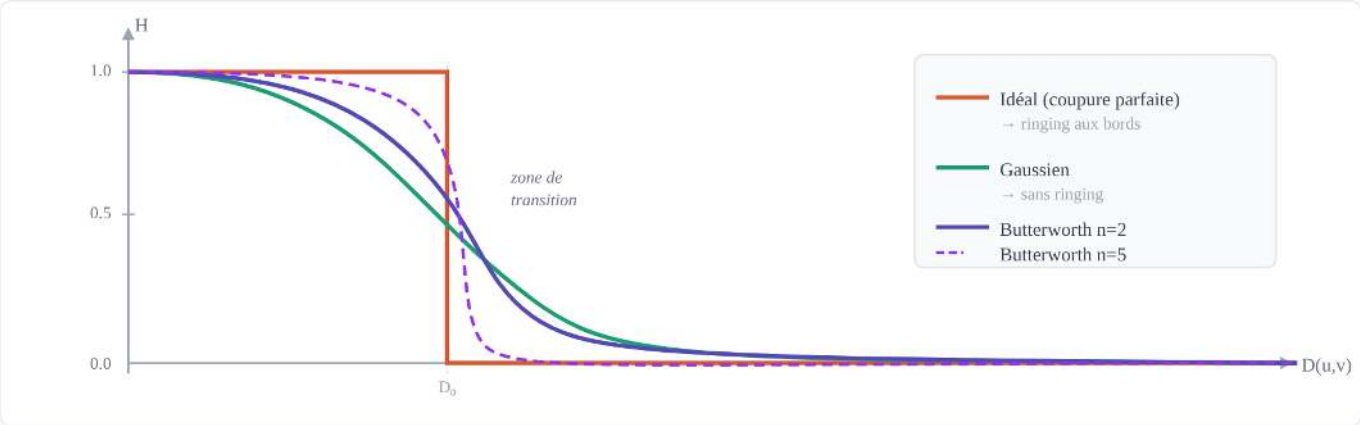


Figure 6.11: Filtros passe-bas.

6.5.2 Filtres Passe-Haut et Passe-Bande

Les filtres passe-haut peuvent être obtenus à partir d'un filtre passe-bas complémentaire, défini comme :

$$H_{\text{HP}}(u,v) = 1 - H_{\text{LP}}(u,v)$$

Ce type de filtre préserve les composantes haute fréquence, en mettant en évidence les contours et les détails, tout en atténuant les régions de variation lente.

Les filtres passe-bande préservent uniquement une bande intermédiaire de fréquences, limitée par deux rayons D_L et D_H :

$$H_{\mathrm{BP}}(u,v) = H_{\mathrm{LP}}^{(D_H)}(u,v) \cdot \left[1 - H_{\mathrm{LP}}^{(D_L)}(u,v)\right]$$

Ce type de filtrage est utile lorsque l'on souhaite supprimer simultanément les composantes basse et haute fréquence, en ne préservant que les structures d'échelle intermédiaire.

Une application importante est la suppression du **bruit périodique**, dans lequel des motifs réguliers apparaissent comme des pics localisés dans le spectre de magnitude. Ces pics peuvent être atténués à l'aide de filtres *notch* (réjecteur de bande), positionnés spécifiquement sur les fréquences indésirables.

Des exemples de filtres dans le domaine fréquentiel sont présentés dans le simulateur de la Figure 6.12, Figure 6.13 et Figure 6.14..

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap05/sim-05-filtros.html

Figure 6.12: Simulateur interactif de filtres dans le domaine fréquentiel.

```python
import io

def distancia_centro(M, N):
    """Matrice des distances au centre du spectre."""
    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):
    """Applique le filtre H (centré) à l'image via FFT."""
    F  = np.fft.fftshift(np.fft.fft2(img.astype(np.float64)))
    Fg = F * H
    g  = np.real(np.fft.ifft2(np.fft.ifftshift(Fg)))
    return cv2.normalize(g, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)


M, N = img_gray.shape
D    = distancia_centro(M, N)
D0   = 30      # fréquence de coupure
n_bw = 2       # ordre du Butterworth

#   Fonctions de transfert
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))

#   Images filtrées
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)

#   Profils 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)

#   Filtres H visualisés
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 idéal", "LPF gaussien", "LPF Butterworth (n=2)",
        "H idéal",  "H gaussien","H Butterworth", "Profils H(u,v)"
    ],
    cols=4, figsize=(16, 9)
)
```

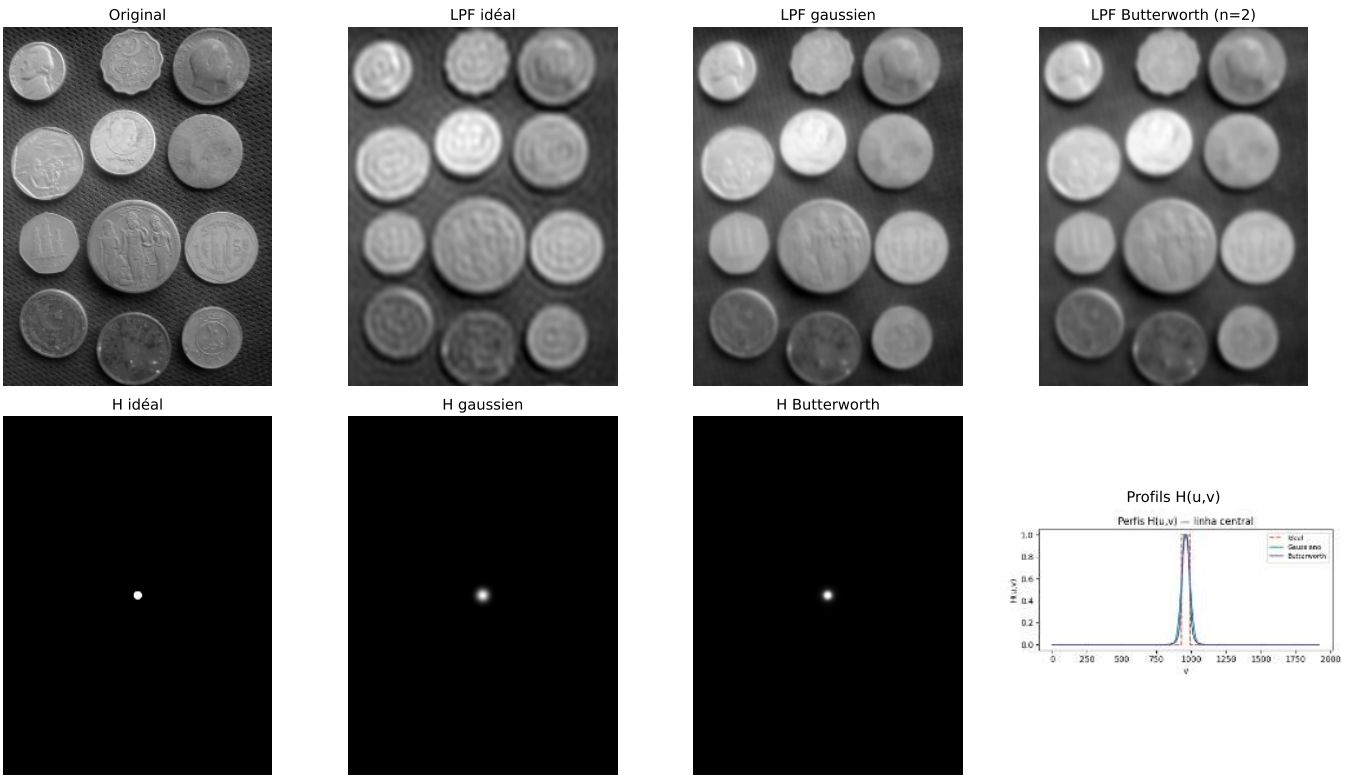


Figure 6.13: Comparaison entre filtres passe-bas : idéal (D =30), gaussien (D =30) et Butterworth (D =30, n=2). Profils de H(u,v) le long d'une ligne centrale et images filtrées correspondantes.

```
# Filtre passe-haut : complément du passe-bas gaussien
# Réutilise aplicar_filtro_freq() définie dans la cellule précédente
H_alta   = 1 - H_gauss
img_alta = aplicar_filtro_freq(img_gray, H_alta)

mm.show(
    [img_gray, img_alta],
    titles=["Original", "Passe-haut gaussien ($D_0=30$)"],
    cols=2
)
```

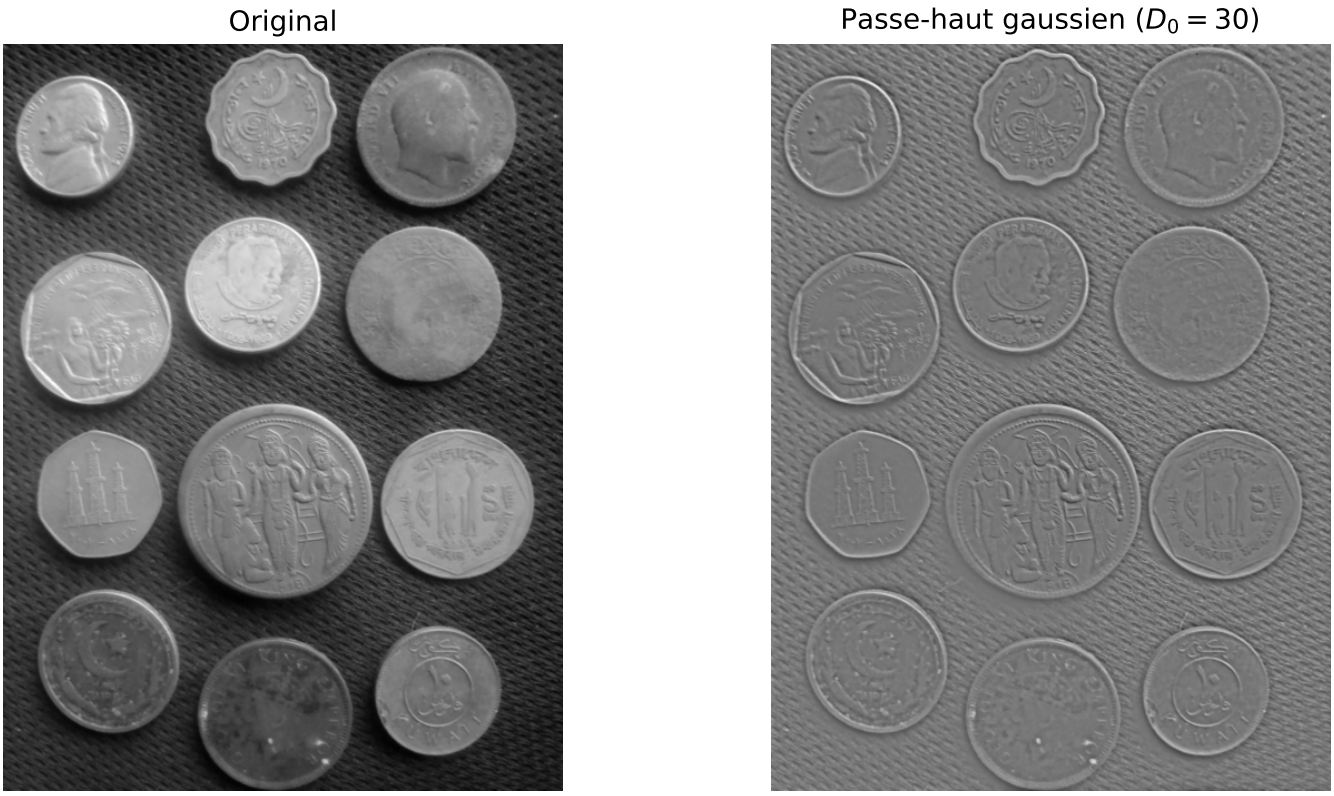


Figure 6.14: Filtre passe-haut gaussien. (a) Original ; (b) Filtre passe-haut (D =30) - les bords des pièces et le fond texturé sont mis en valeur.

6.5.3 Suppression du bruit périodique

Le bruit périodique — associé à des interférences électriques, à des motifs réguliers de capteurs ou à des artefacts de numérisation — apparaît dans le spectre de Fourier sous forme de **pics ponctuels symétriques autour du centre**.

Le filtre **rejette-bande (*notch*)** atténue sélectivement ces fréquences, tout en préservant les autres composantes de l'image. Un exemple d'application est présenté dans la Figure 6.15..

```
#   Image avec bruit périodique synthétique
h_img, w_img = img_gray.shape
x = np.arange(w_img)
y = np.arange(h_img)
X, Y = np.meshgrid(x, y)

# Utiliser des fréquences entières et cohérentes (important pour une restauration parfaite)
u0, v0 = 20, 20  # fréquences exactes du bruit

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)

#   Spectre de l'image bruitée
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)

#   Masque notch
mascara = np.ones((h_img, w_img), dtype=np.float64)
r_notch = 8  # rayon du notch (ajustement fin si nécessaire)
```

```python
def suprimir_pico(mask, cy, cx, r):
    """Met à zéro un disque de rayon r centré 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

# Coordonnées centrales
cy, cx = h_img // 2, w_img // 2

# Supprimer les 4 pics symétriques (important !)
for dy, dx in [(v0, u0), (-v0, -u0), (v0, -u0), (-v0, u0)]:
    mascara = suprimir_pico(mascara, cy + dy, cx + dx, r_notch)

mascara_vis = (mascara * 255).astype(np.uint8)

#   Filtrage et reconstruction
F_filtrada = F_r * mascara
img_rest = np.real(np.fft.ifft2(np.fft.ifftshift(F_filtrada)))
img_rest_vis = cv2.normalize(img_rest, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

# Évaluation
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 restaurée) : {psnr:.2f} dB")

#   Visualisation
mm.show(
    [img_ruidosa, mag_vis, mascara_vis, img_rest_vis],
    titles=[
        "Avec bruit périodique",
        "Spectre (log)",
        "Masque notch",
        f"Restaurée (PSNR={psnr:.1f} dB)"
    ],
    cols=4,
    figsize=(16, 4)
)
```

```
PSNR (original vs restaurée) : 32.93 dB
```

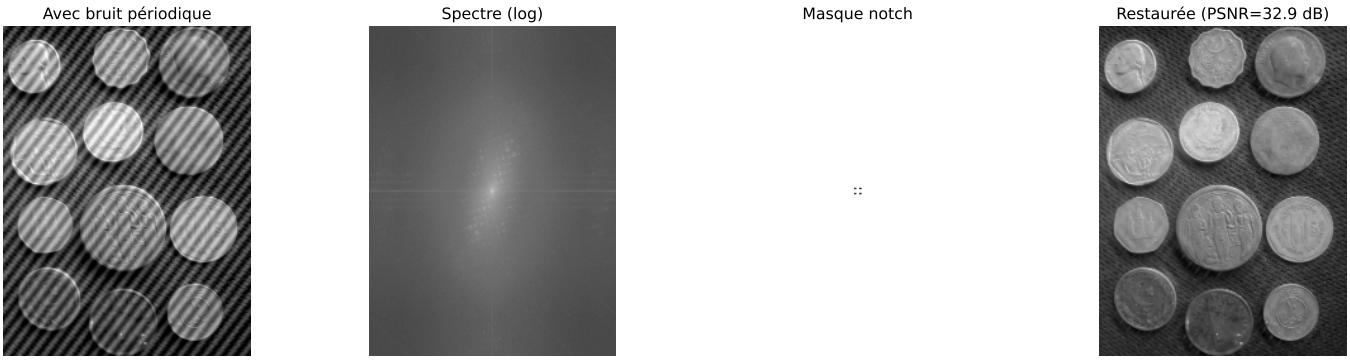


Figure 6.15: Suppression du bruit périodique via filtre *notch* dans le domaine fréquentiel : (a) image avec bruit sinusoïdal, (b) spectre montrant les pics de bruit, (c) masque *notch* centré sur les pics, (d) image restaurée.

📌 Synthèse — Filtres spectraux

Filtre	Effet visuel	Artefact	Utilisation
Passe-bas idéal	Lissage intense	*Ring*	Illustratif
Passe-bas gaussien	Lissage doux	N'présente pas de *ring*	Lissage général
Passe-bas de Butterworth	Lissage contrôlé	*Ring* (ordres élevés)	Compromis entre lissage et sélectivité
Passe-haut	Renforcement des contours	Amplification du bruit	Détection des contours
Notch	Suppression sélective de fréquences	Distorsions locales possibles	Suppression du bruit périodique

La conception de filtres dans le domaine fréquentiel consiste à définir des masques spectraux. Cependant, des effets dans le domaine spatial, tels que le *ring* et le flou, émergent directement de ces choix dans le spectre.

6.6 *Ondelettes* et Multirésolution

La Transformée de Fourier décompose le signal en fréquences **globales** : chaque coefficient $F(u, v)$ reçoit des contributions de toute l'image, sans information explicite sur la localisation spatiale de ces fréquences. Ainsi, les structures localisées, comme les bords, sont représentées de manière distribuée dans le spectre.

Les ***ondelettes*** **(ondelettes)** surmontent cette limitation en utilisant des fonctions de base **localisées dans l'espace**, qui peuvent être déplacées et mises à l'échelle. Ces fonctions possèdent un **support compact**, c'est-à-dire qu'elles sont différentes de zéro uniquement dans une région finie du domaine, permettant une représentation simultanée en termes de **fréquence et de localisation spatiale**.

6.6.1 La Limite de la Transformée de Fourier : localisation spatiale

La Transformée de Fourier décrit avec précision **quelles fréquences sont présentes** dans un signal, mais ne représente pas explicitement **où ces fréquences se produisent dans l'espace**.

Dans l'expérience présentée dans la Figure 6.16, deux images avec des structures localisées à des positions différentes produisent des spectres de magnitude pratiquement identiques. Cela est dû au fait que la représentation de Fourier est globale : chaque coefficient reçoit une contribution de l'ensemble de l'image.

Par conséquent, le spectre de magnitude ne représente pas explicitement la localisation des contours ou d'autres structures, mais uniquement la distribution des fréquences présentes. Cette limitation a motivé le développement de représentations multirésolution, telles que la Transformée *Wavelet* Discrète (DWT), capables de décrire simultanément la fréquence et la localisation spatiale des structures de l'image.

```
img_sinal1 = np.zeros((128, 128)); img_sinal1[:, 20:25] = 1; img_sinal1[100:105, :] = 1
img_sinal2 = np.zeros((128, 128)); img_sinal2[:, 90:95] = 1; img_sinal2[30:35, :] = 1

mag1 = np.log1p(np.abs(np.fft.fftshift(np.fft.fft2(img_sinal1))))
mag2 = np.log1p(np.abs(np.fft.fftshift(np.fft.fft2(img_sinal2))))

mm.show([img_sinal1, mag1, img_sinal2, mag2],
        titles=["Signal A", "Spectre A", "Signal B (Décalé)", "Spectre B"],
        cols=4, figsize=(14, 4)
        )
```

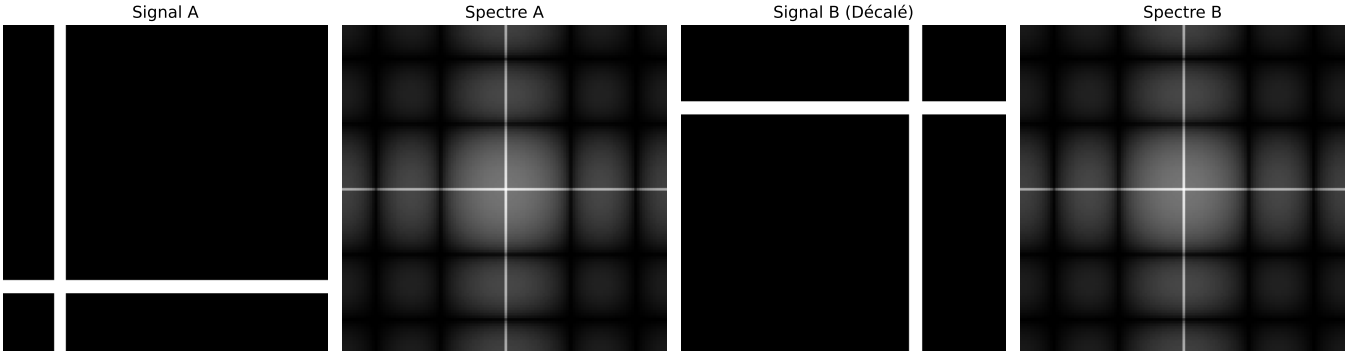


Figure 6.16: La transformée de Fourier globale est aveugle à la position. Les spectres ne disent pas où se trouvent les bords.

6.6.2 Transformée *Wavelet* Discrète 2D

La Transformée *Wavelet* Discrète (DWT) applique, séparément dans les directions horizontale et verticale, deux filtres complémentaires : un **passe-bas** h (approximation) et un **passe-haut** g (détails), suivis d'un sous-échantillonnage par un facteur de 2 dans chaque dimension. Ce processus produit quatre sous-bandes, dont les noms indiquent la combinaison des filtres appliqués dans chaque direction (L = *Low-pass*, passe-bas ; H = *High-pass*, passe-haut). Les caractéristiques de chaque sous-bande sont résumées dans la Table 6.3.

$$\mathrm{DWT}(f) = \{ \underbrace{\mathrm{LL}}_{\text{approx.}} , \underbrace{\mathrm{LH}}_{\text{détails horizontaux}} , \underbrace{\mathrm{HL}}_{\text{détails verticaux}} , \underbrace{\mathrm{HH}}_{\text{détails diagonaux}} \}.$$

Tableau 6.3: Sous-bandes produites par la Transformée *Wavelet* Discrète 2D (DWT), indiquant les filtres appliqués dans chaque direction et le contenu prédominant de chaque composante.

Sous-bande	Filtres appliqués	Contenu visuel
LL	bas × bas	Approximation de l'image (version lissée et réduite)
LH	bas × haut	Bords horizontaux et variations verticales
HL	haut × bas	Bords verticaux et variations horizontales
HH	haut × haut	Détails diagonaux et textures

La décomposition peut être appliquée récursivement sur la sous-bande LL, générant une représentation multi-résolution. Après J niveaux, on obtient une structure avec $3J + 1$ sous-bandes, où chaque nouveau niveau réduit la résolution de la composante d'approximation.

> **Lien avec les CNN**
>
> La décomposition multi-résolution des *wavelets* possède une relation conceptuelle avec les représentations hiérarchiques utilisées dans les réseaux de neurones convolutifs (CNN). Dans les deux cas, des étapes successives de filtrage et de réduction de résolution produisent des descriptions de plus en plus abstraites de l'image. Cependant, les ***wavelets* utilisent des filtres mathématiquement définis et reconstructibles**, tandis que les **CNN apprennent leurs filtres pendant l'entraînement**.

6.6.3 Familles d'*ondelettes*

Différentes familles d'*ondelettes* présentent des compromis distincts entre **support spatial**, régularité et capacité de compression. Le **support** correspond à l'étendue de la fonction *ondelette* dans le domaine spatial

: plus le support est petit, plus la fonction est localisée ; plus il est grand, plus sa représentation tend à être régulière, mais avec un coût de calcul plus élevé. La Table 6.4 compare certaines des familles les plus utilisées.

Tableau 6.4: Comparaison entre familles d'*ondelettes*, mettant en évidence la longueur du support, le nombre de moments nuls, la symétrie et les applications typiques.

Ondelette	Longueur du support	Moments nuls	Symétrie	Usage typique
Haar	2	1	Asymétrique	Introduction et analyse de base
Daubechies db4	8	4	Asymétrique	Compression et analyse générale
Symlet sym4	8	4	Quasi symétrique	Reconstruction de signaux
Biorthogonale 5/3	5/3	2/2	Symétrique	JPEG 2000 sans perte
Biorthogonale 9/7	9/7	4/4	Symétrique	JPEG 2000 avec perte

Les **moments nuls** mesurent la capacité de l'*ondelette* à représenter les régions lisses de l'image avec peu de coefficients non nuls. Une *ondelette* à p moments nuls annule exactement les polynômes de degré jusqu'à $p-1$. Par conséquent, plus le nombre de moments nuls est élevé, plus l'efficacité de compression tend à être grande dans les régions homogènes, bien que cela implique généralement des fonctions à support plus long.

La Figure 6.17 présente les fonctions de base (*ondelettes*) $\psi(t)$ dans le domaine spatial. Ces fonctions possèdent un **support compact**, c'est-à-dire qu'elles sont non nulles uniquement sur une région finie du domaine, contrairement aux sinusoïdes de la Transformée de Fourier, qui s'étendent sur tout le domaine.

```
#   Importation de pywt
try:
    import pywt
except ImportError:
    import subprocess
    subprocess.run(["pip", "install", "PyWavelets", "-q"])
    import pywt

wavelet_haar = pywt.Wavelet('haar')
wavelet_db4  = pywt.Wavelet('db4')

phi_h, psi_h, x_h = wavelet_haar.wavefun(level=4)
phi_d, psi_d, x_d = wavelet_db4.wavefun(level=4)

fig, ax = plt.subplots(1, 2, figsize=(10, 3))
ax[0].plot(x_h, psi_h, 'b', lw=2); ax[0].set_title("Ondaleta Haar ( )")
ax[1].plot(x_d, psi_d, 'g', lw=2); ax[1].set_title("Ondaleta Daubechies 4 ( )")
plt.tight_layout(); plt.show()
```

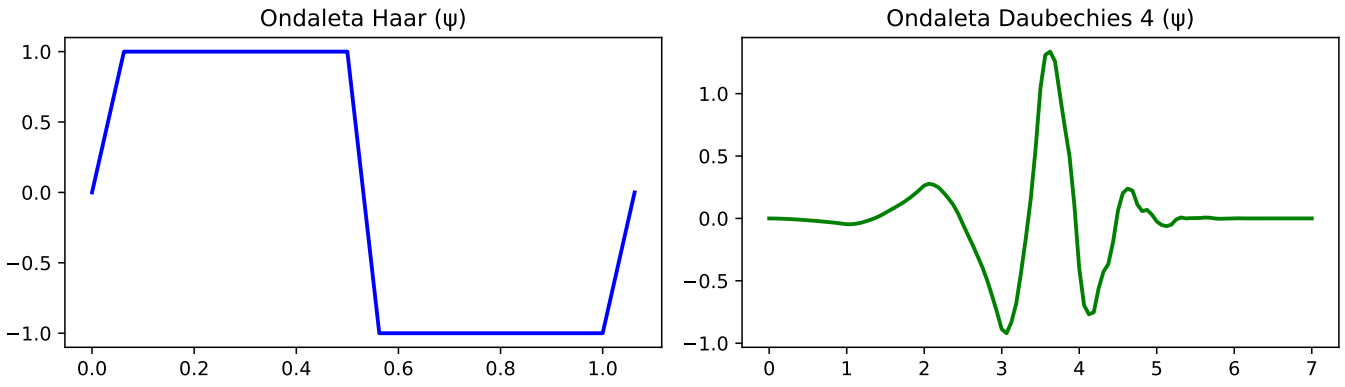


Figure 6.17: Fonctions de la *wavelet* ( ). Notez comme elles décroissent rapidement vers zéro (support compact), contrairement aux sinusoïdes infinies de Fourier.

Le diagramme de la Figure 6.18 illustre l'analyse multirésolution effectuée par la DWT, dans laquelle la sous-bande d'approximation (LL) est successivement décomposée, formant une représentation hiérarchique à deux niveaux.

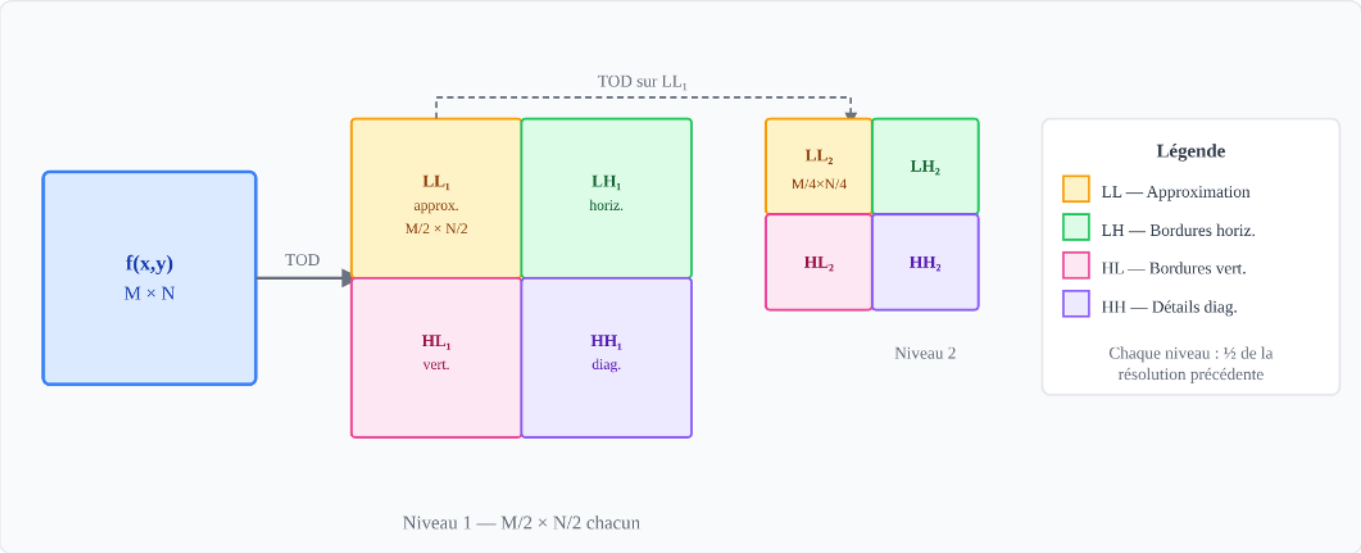


Figure 6.18: Schéma de la décomposition *wavelet* 2D en deux niveaux.

Le simulateur de la Figure 6.19 permet d'explorer de manière interactive la Transformée *Wavelet* Discrète 2D (TWD) à l'aide de la *wavelet* de Haar. La décomposition en sous-bandes met en évidence la séparation entre la composante d'approximation et les composantes de détail de l'image.

Les différents motifs d'entrée permettent d'observer le comportement directionnel des filtres. Dans les images comportant des bords horizontaux et verticaux, les sous-bandes LH et HL mettent en évidence, respectivement, les variations verticales et horizontales de l'intensité. Dans les régions à variation douce, la majeure partie de l'énergie se concentre dans la sous-bande d'approximation LL, tandis que les sous-bandes de détail présentent des coefficients proches de zéro.

L'analyse multirésolution peut également être observée en augmentant le nombre de niveaux de décomposition. Dans ce cas, seule la sous-bande LL_1 est à nouveau décomposée, donnant naissance aux sous-bandes LL_2, LH_2, HL_2 et HH_2, qui forment le deuxième niveau de la représentation hiérarchique.

Dans les motifs constitués de régions homogènes de grande étendue, comme un dégradé doux ou un damier composé de grands blocs, l'énergie demeure principalement concentrée dans la sous-bande LL. Dans le dégradé, cela s'explique par le fait que les différences entre pixels voisins sont faibles. Dans le damier, en revanche, les pixels possèdent pratiquement la même intensité à l'intérieur de chaque bloc, de sorte que seules

les frontières entre les blocs produisent des coefficients non nuls dans les sous-bandes de détail. Comme ces frontières n'occupent qu'une petite fraction de l'image, leur contribution à l'énergie totale reste réduite.

Pour permettre l'analyse visuelle de ces variations subtiles, le simulateur intègre un contrôle de gain de contraste des détails (variant de 1 à 8). Ce paramètre agit comme un facteur d'amplification linéaire appliqué exclusivement aux coefficients des sous-bandes de détail (LH, HL et HH) avant leur rendu à l'écran. Dans les scénarios de transition douce (comme le dégradé) ou d'uniformité locale (comme l'intérieur des blocs du damier), les différences numériques calculées par le filtre passe-haut de Haar donnent des coefficients très proches de zéro, ce qui rendrait les quadrants correspondants sombres et imperceptibles à l'œil nu. En multipliant ces valeurs par le gain, le simulateur fait ressortir visuellement les structures de haute fréquence cachées et met en évidence l'orientation des bords restants.

Le graphique de l'énergie par sous-bande quantifie cette répartition entre la composante d'approximation et les composantes de détail, démontrant que le gain visuel ne modifie pas la métrique originale de l'énergie. Dans les images naturelles, la majeure partie de l'énergie se concentre dans la sous-bande LL, tandis que les sous-bandes LH, HL et HH représentent principalement les bords, les textures et d'autres variations locales de l'intensité.

6.6.4 Analyse multirésolution avec la DWT 2D

La transformée *wavelet* discrète 2D (DWT) décompose une image en composantes d'approximation et de détail, organisées de manière hiérarchique à différentes échelles et orientations. Comme les sous-bandes de détail dans les images naturelles présentent souvent des coefficients de faible contraste, les exemples pratiques suivants utilisent un motif géométrique synthétique généré en Python. Cette approche reproduit le comportement du simulateur Figure 6.19, rendant visuellement explicites les effets du filtrage spatial et de la décomposition multirésolution.

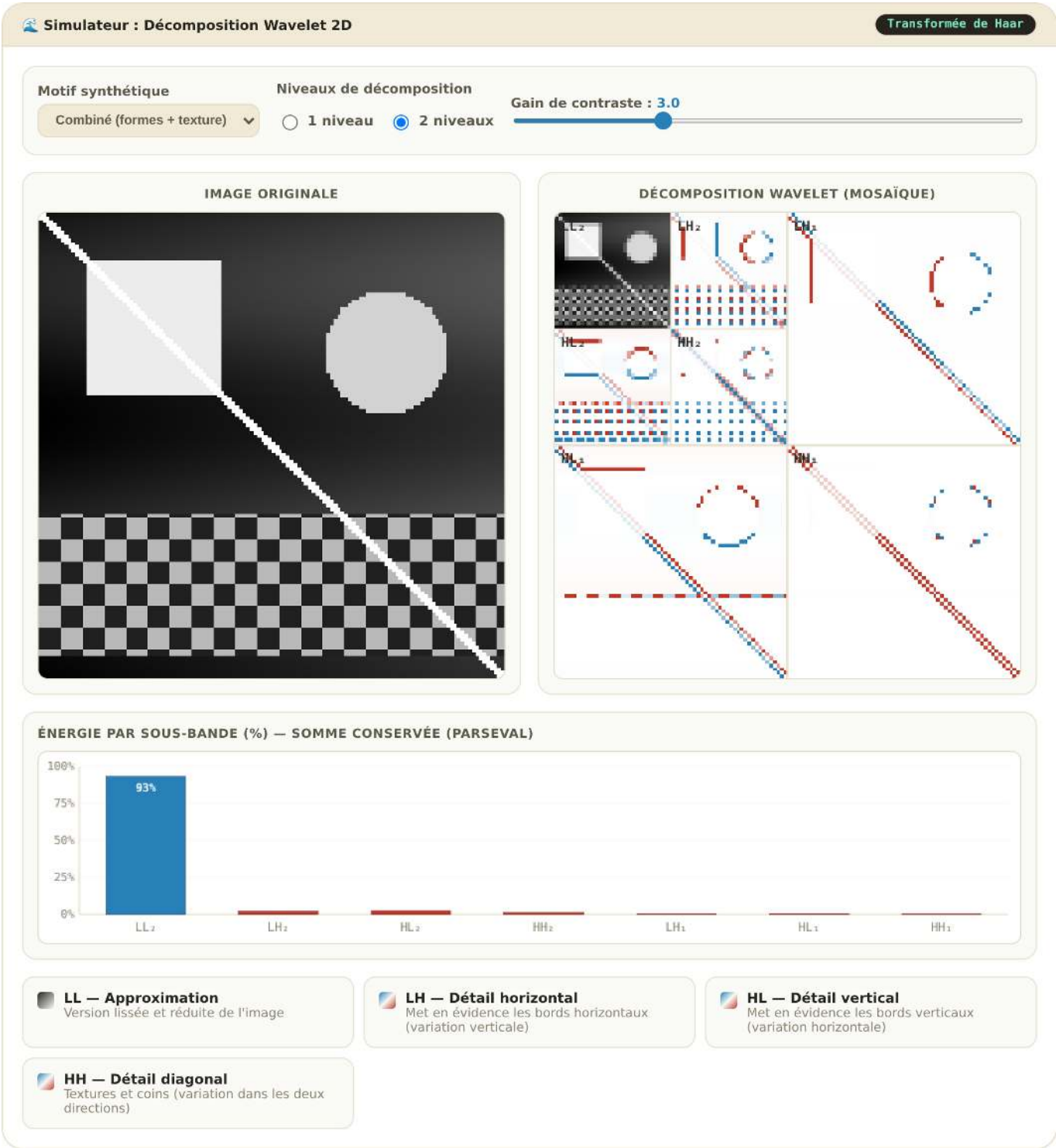
6.6.4.1 Décomposition en Mosaïque à Multiples Niveaux

La Figure 6.20 illustre la structure hiérarchique de la DWT à deux niveaux en utilisant la *wavelet* de Haar. Le processus repose sur l'application combinée de filtres passe-bas et passe-haut dans les directions horizontale et verticale, suivis d'un sous-échantillonnage par un facteur de 2.

Au premier niveau, l'image originale donne naissance à la sous-bande d'approximation (LL_1) ainsi qu'aux composantes de détail horizontale (LH_1), verticale (HL_1) et diagonale (HH_1). Dans l'analyse multirésolution, la sous-bande LL_1 est à nouveau filtrée et sous-échantillonnée, générant le deuxième niveau de décomposition (LL_2, LH_2, HL_2 et HH_2).

Pour faciliter l'interprétation visuelle des composantes de détail, le code extrait la valeur absolue de leurs coefficients et applique une normalisation linéaire (*min-max*) afin d'occuper toute la plage dynamique des niveaux de gris [0, 255]. Cette opération transforme les régions homogènes (coefficients nuls) en noir et met en évidence en blanc les contours et textures extraits à chaque échelle et orientation.

```
try:
    import pywt
    HAS_PYWT = True
except ImportError:
    import subprocess
    subprocess.run(["pip", "install", "PyWavelets", "-q"])
    import pywt
    HAS_PYWT = True

import numpy as np
import cv2

#   Génération de l'Image Synthétique (Même motif 'combined' du simulateur)
def gerar_imagem_sintetica(N=256):
    img = np.zeros((N, N), dtype=np.float64)
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap05/sim-05-wavelet.html

Figure 6.19: Simulation de la décomposition *wavelet* 2D.

```
    for y in range(N):
        for x in range(N):
            v = 55 + 35 * (x / N) + 15 * np.sin(y / 24)
            # Carré
            if 24 < x < 100 and 24 < y < 100:
                v = 225
            # Cercle
            cx, cy, r = 190, 76, 34
            if (x - cx)**2 + (y - cy)**2 < r**2:
                v = 205
            # Texture périodique (inférieure)
            if y > 164 and y < 244:
                p = 12
                v = 185 if ((x // p + y // p) % 2 == 0) else 65
            # Ligne diagonale
            if abs(x - y) < 4:
                v = 240
            img[y, x] = np.clip(v, 0, 255)
    return img.astype(np.uint8)

# Remplace l'image sombre de pièces par le motif synthétique clair
img_gray = gerar_imagem_sintetica(256)

#   Décomposition wavelet à 2 niveaux
wavelet   = "haar"
img_float = img_gray.astype(np.float64)

# Niveau 1
coefs1 = pywt.dwt2(img_float, wavelet)
LL1, (LH1, HL1, HH1) = coefs1

# Niveau 2 (appliqué sur LL1)
coefs2 = pywt.dwt2(LL1, wavelet)
LL2, (LH2, HL2, HH2) = coefs2

def sb_vis(sb):
    """Normalise la sous-bande pour la visualisation [0,255]."""
    return cv2.normalize(np.abs(sb), None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

print(f"Forme originale     : {img_gray.shape}")
print(f"LL1 (niveau 1)      : {LL1.shape}  |  LH1/HL1/HH1 : {LH1.shape}")
print(f"LL2 (niveau 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))
```

```
Forme originale     : (256, 256)
LL1 (niveau 1)      : (128, 128)  |  LH1/HL1/HH1 : (128, 128)
LL2 (niveau 2)      : (64, 64)    |  LH2/HL2/HH2 : (64, 64)
```

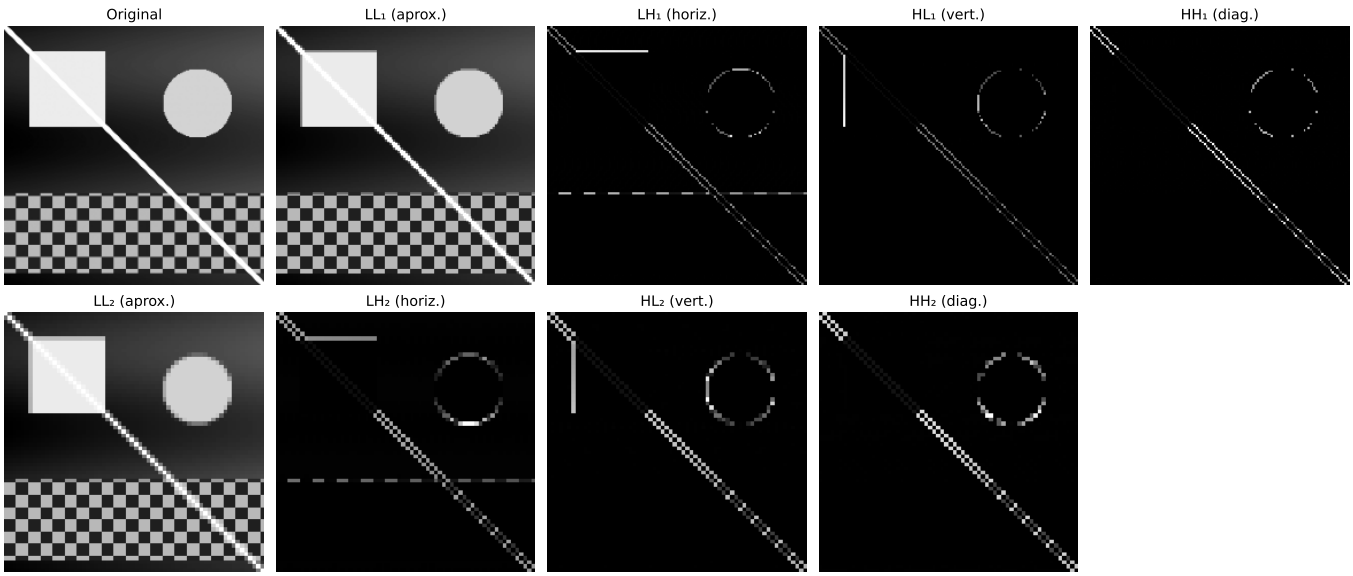


Figure 6.20: Décomposition *wavelet* 2D à 2 niveaux avec *wavelet* de Haar : sous-bandes LL, LH, HL, HH à chaque niveau. Les sous-bandes de détail révèlent des structures orientées à différentes échelles en utilisant un motif synthétique.

6.6.4.2 Le Compromis entre Localisation et Régularité

Le choix de la fonction de base (*ondelette*) influence directement la manière dont les caractéristiques de l'image sont distribuées et codées par les coefficients de la DWT. La Figure 6.21 compare les résultats pratiques obtenus en appliquant quatre familles distinctes sur le motif géométrique synthétique : `haar`, `db4`, `sym4` et `bior2.2`.

En raison de son support court et de sa forme en fonction échelon, l'ondelette de Haar produit des coefficients hautement localisés au niveau des discontinuités spatiales, générant des bords fins et nets dans les sous-bandes de détail. En revanche, des familles comme Daubechies (`db4`) et Symlets (`sym4`), qui présentent un support plus étendu (filtres plus longs) et un plus grand nombre de moments nuls, génèrent des réponses plus lisses et plus distribuées autour des transitions, ce qui peut introduire de légères oscillations ou un adoucissement aux frontières abruptes.

Ce comportement met en évidence le compromis classique (*trade-off*) de l'analyse multirésolution : des supports plus petits favorisent la localisation spatiale exacte des bords, tandis que des supports plus grands et un nombre plus élevé de moments nuls tendent à produire des représentations plus parcimonieuses et plus régulières. Cette régularité et cette capacité d'atténuation des hautes fréquences garantissent une plus grande efficacité dans la compaction de l'énergie, des caractéristiques fondamentales pour les applications de compression de données et de débruitage (*denoising*).

```
import numpy as np
import cv2
import pywt

# Garantit que img_gray et img_float utilisent le même motif synthétique clair
if 'gerar_imagem_sintetica' in globals():
    img_gray = gerar_imagem_sintetica(256)
else:
    # Repli au cas où le bloc précédent n'aurait pas été exécuté dans la même session
    def gerar_imagem_sintetica(N=256):
        img = np.zeros((N, N), dtype=np.float64)
        for y in range(N):
            for x in range(N):
                v = 55 + 35 * (x / N) + 15 * np.sin(y / 24)
```

```
                if 24 < x < 100 and 24 < y < 100: v = 225
                cx, cy, r = 190, 76, 34
                if (x - cx)**2 + (y - cy)**2 < r**2: v = 205
                if y > 164 and y < 244:
                    p = 12
                    v = 185 if ((x // p + y // p) % 2 == 0) else 65
                if abs(x - y) < 4: v = 240
                img[y, x] = np.clip(v, 0, 255)
        return img.astype(np.uint8)
    img_gray = gerar_imagem_sintetica(256)

img_float = img_gray.astype(np.float64)

wavelets_comp = ["haar", "db4", "sym4", "bior2.2"]
imgs_comp, titles_comp = [], []

for wname in wavelets_comp:
    LL, (LH, HL, HH) = pywt.dwt2(img_float, wname)
    imgs_comp  += [sb_vis(LL), sb_vis(HH)]
    titles_comp += [f"{wname} - LL ", f"{wname} - HH "]

mm.show(imgs_comp, titles=titles_comp, cols=4, figsize=(14, 8))
```

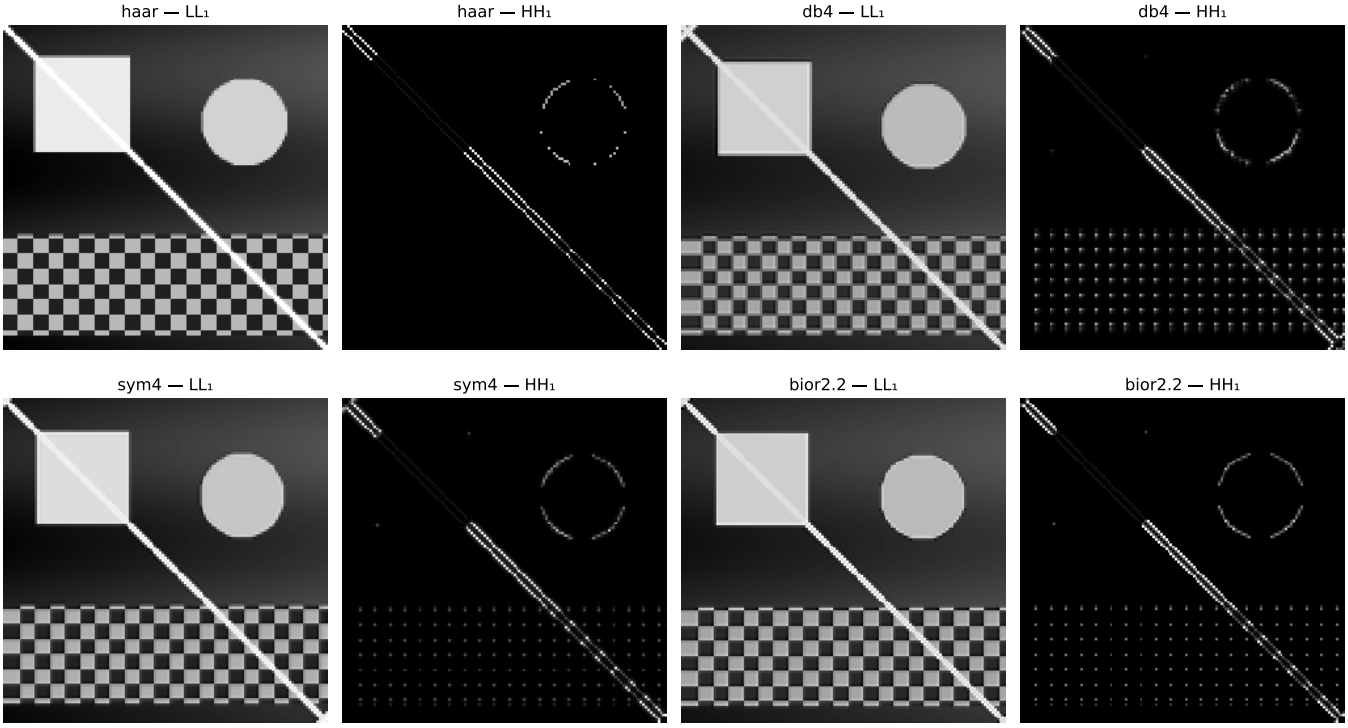


Figure 6.21: Comparaison entre famílias de *ondelettes* : Haar, db4, sym4 et bior2.2. Sous-bande LL (approximation) et HH (diagonale) pour chaque choix, illustrant le compromis entre compactage et douceur sur la base du motif synthétique.

6.6.4.3 Limiarização de Coeficientes e Compressão

Uma das principais aplicações da Transformada *Wavelet* Discreta (DWT) é a compressão de dados, impulsionada pela capacidade de representação **esparsa** dos coeficientes. A Figure 6.22 ilustra o efeito da limiarização abrupta (*hard thresholding*), técnica na qual coeficientes de detalhe com magnitude inferior a um limiar T são integralmente anulados antes do processo de síntese realizado pela Transformada *Wavelet* Discreta Inversa (IDWT).

À medida que o limiar T é elevado, um volume crescente de coeficientes de alta frequência é zerado. Por concentrarem menor energia, a remoção dessas componentes reduz consideravelmente a quantidade de informação necessária para representar a imagem, mantendo a componente de aproximação global (a subbanda LL mais profunda) intacta para preservar a estrutura macro. Visualmente, esse descarte de coeficientes manifesta-se através do desaparecimento progressivo de texturas finas e da suavização de transições abruptas de intensidade.

A fidelidade da imagem reconstruída frente à original é quantificada pela métrica de **Pico da Relação Sinal-Ruído (PSNR, *Peak Signal-to-Noise Ratio*)**, expressa em decibéis (dB). Valores mais altos de PSNR indicam menor distorção e maior proximidade matemática com o sinal original. O experimento prático evidencia o decaimento gradual do PSNR conforme a agressividade da limiarização aumenta, permitindo avaliar numericamente o limiar ótimo para o balanço entre compressão e degradação visual.

6.6.4.4 Seuil des Coefficients et Compression

L'une des principales applications de la Transformée *Wavelet* Discrète (DWT) est la compression de données, portée par la capacité de représentation **parcimonieuse** des coefficients. La Figure 6.22 illustre l'effet du seuillage abrupt (*hard thresholding*), technique dans laquelle les coefficients de détail dont la magnitude est inférieure à un seuil T sont intégralement annulés avant le processus de synthèse réalisé par la Transformée *Wavelet* Discrète Inverse (IDWT).

À mesure que le seuil T est augmenté, un volume croissant de coefficients haute fréquence est mis à zéro. Comme ils concentrent moins d'énergie, la suppression de ces composantes réduit considérablement la quantité d'information nécessaire pour représenter l'image, tout en maintenant intacte la composante d'approximation globale (la sous-bande LL la plus profonde) afin de préserver la structure macroscopique. Visuellement, ce rejet de coefficients se manifeste par la disparition progressive des textures fines et par l'adoucissement des transitions abruptes d'intensité.

La fidélité de l'image reconstruite par rapport à l'originale est quantifiée par la métrique du **Pic du Rapport Signal sur Bruit (PSNR, *Peak Signal-to-Noise Ratio*)**, exprimée en décibels (dB). Des valeurs plus élevées de PSNR indiquent une distorsion moindre et une plus grande proximité mathématique avec le signal original. L'expérience pratique met en évidence la décroissance progressive du PSNR à mesure que l'agressivité du seuillage augmente, permettant d'évaluer numériquement le seuil optimal pour l'équilibre entre compression et dégradation visuelle.

```
import numpy as np
import cv2
import pywt

# Garantit que img_gray utilise le même motif synthétique clair
if 'gerar_imagem_sintetica' in globals():
    img_gray = gerar_imagem_sintetica(256)
else:
    # Repli au cas où le bloc précédent n'a pas été exécuté dans la même session
    def gerar_imagem_sintetica(N=256):
        img = np.zeros((N, N), dtype=np.float64)
        for y in range(N):
            for x in range(N):
                v = 55 + 35 * (x / N) + 15 * np.sin(y / 24)
                if 24 < x < 100 and 24 < y < 100: v = 225
                cx, cy, r = 190, 76, 34
                if (x - cx)**2 + (y - cy)**2 < r**2: v = 205
                if y > 164 and y < 244:
                    p = 12
                    v = 185 if ((x // p + y // p) % 2 == 0) else 65
                if abs(x - y) < 4: v = 240
                img[y, x] = np.clip(v, 0, 255)
        return img.astype(np.uint8)
```

```
    img_gray = gerar_imagem_sintetica(256)

def dwt_threshold_reconstruct(img, wavelet='db4', nivel=2, threshold=0.0):
    """Décompose, applique le seuil et reconstruit via IDWT."""
    coefs = pywt.wavedec2(img.astype(np.float64), wavelet, level=nivel)
    # Copie et applique le hard thresholding sur tous les détails
    coefs_t = [coefs[0]]   # LL final n'est pas seuillé
    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)
    # Recadrage à la dimension originale
    rec = rec[:img.shape[0], :img.shape[1]]
    return np.clip(rec, 0, 255).astype(np.uint8)

thresholds = [0, 10, 30, 60, 100]
imgs_thr   = [img_gray]
titles_thr = ["Original"]

for t in thresholds:
    rec  = dwt_threshold_reconstruct(img_gray, threshold=t)
    psnr = cv2.PSNR(img_gray, rec)
    imgs_thr.append(rec)
    titles_thr.append(f"T={t}  PSNR={psnr:.1f} dB")

mm.show(imgs_thr, titles=titles_thr, cols=3, figsize=(14, 10))
```

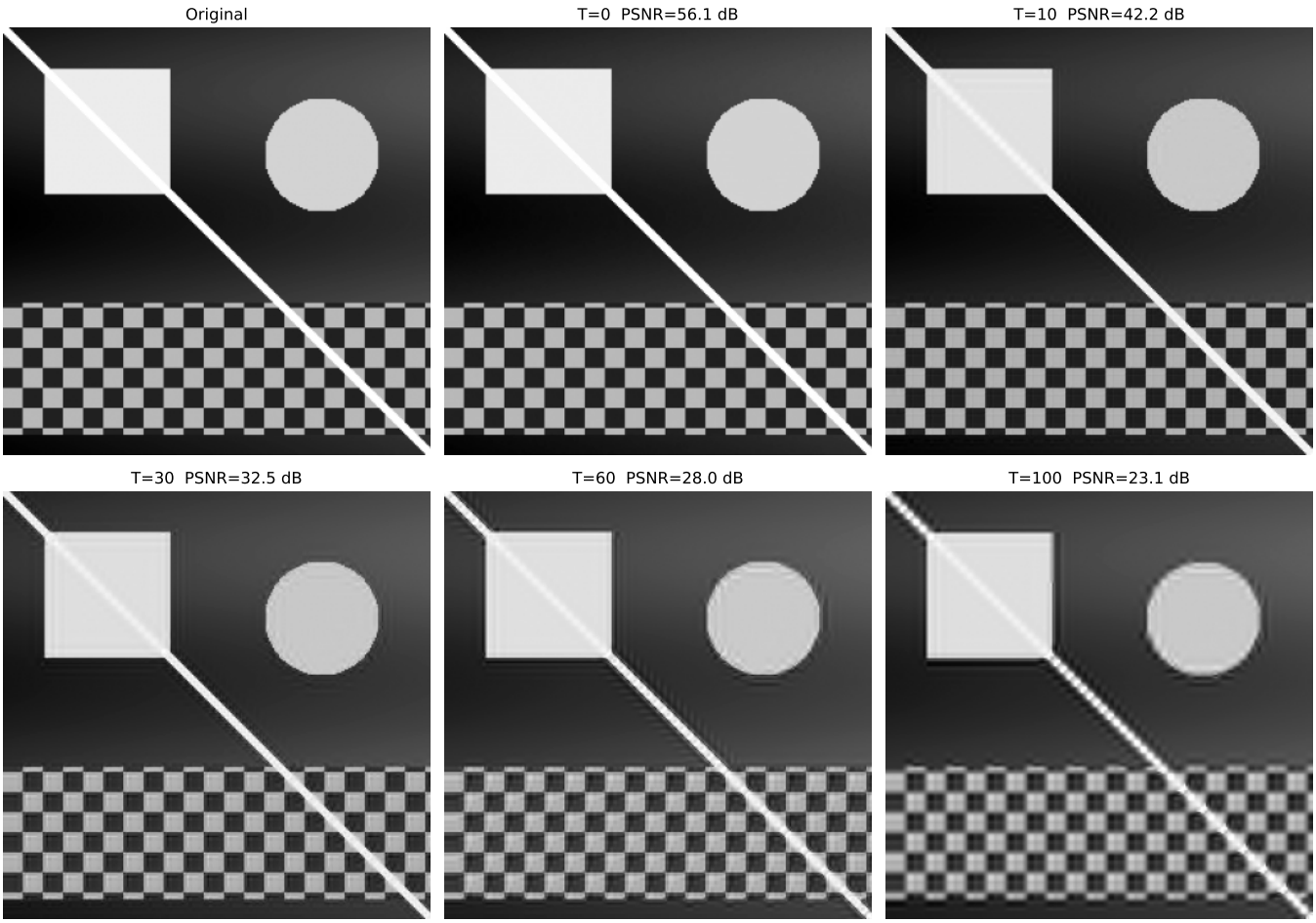


Figure 6.22: Reconstruction *wavelet* avec seuillage des coefficients (*hard thresholding*) : à mesure que le seuil augmente, plus de détails sont mis à zéro, produisant des images progressivement plus lisses. La métrique PSNR quantifie la perte de qualité sur le motif synthétique.

Synthèse — Fourier vs. *wavelets* : quand utiliser chaque approche ?

La Table 6.5 synthétise les principales différences structurelles et opérationnelles entre la transformée de Fourier discrète (TFD) et la transformée en *ondelettes* discrète (TOD).

Tableau 6.5: Comparaison entre la transformée de Fourier discrète (TFD) et la transformée en *ondelettes* discrète (TOD), mettant en évidence leurs principales caractéristiques et applications.

Critère	Fourier (TFD)	*Ondelettes* (TOD)
Fonctions de base	Sinusoïdes à support infini	Fonctions à support compact
Localisation spatiale	Non explicite (globale)	Explicite (locale)
Filtrage spectral	Excellent pour un contrôle fin des fréquences	Basé sur des sous-bandes (échelles)
Compression d'images	Base de la TCD (JPEG traditionnel)	Base de la TOD (JPEG 2000)
Analyse multi-échelle	Non	Oui
Suppression du bruit périodique	Très efficace	Peu indiquée
Signaux non stationnaires	Limitée	Très efficace

En termes pratiques, la TFD s'impose comme l'outil idéal pour l'analyse spectrale pure, la conception de filtres sélectifs dans le domaine fréquentiel et l'atténuation des bruits périodiques et harmoniques. En revanche, la TOD excelle dans les scénarios exigeant une préservation rigoureuse de la localisation spatiale des caractéristiques associée à leur contenu fréquentiel, se distinguant dans la compression de données, l'analyse multirésolution et le traitement des transitions abruptes. Ainsi, les deux transformées doivent être comprises comme des techniques parfaitement complémentaires, traçant des voies distinctes et spécifiques pour la résolution de problèmes en TNI-VC.

ℹ Analogies avec l'audio : limites et précautions

Lorsqu'on établit des analogies entre le traitement d'images et l'audio, il est important de prendre en compte les différences fondamentales :

- Dans les systèmes audio stéréo/multicanaux, la phase entre les canaux est cruciale pour la perception de la localisation spatiale (différences interaurales de phase et de temps).
- Dans les systèmes monauraux, la phase a une influence perceptuelle limitée — l'oreille humaine est relativement insensible à la phase absolue des composantes sinusoïdales isolées.
- Dans les images, la phase de la TFD est toujours fondamentale pour la localisation spatiale des structures, qu'il s'agisse d'une image monochrome ou couleur.

L'analogie entre la phase en audio et la phase en images doit être utilisée avec prudence, en soulignant que, bien que toutes deux portent des informations sur l'organisation spatiale/temporelle du signal, les mécanismes perceptuels sont fondamentalement différents.

6.7 Compression d'images

Alors que les *wavelets* établissent la fondation théorique du standard JPEG 2000, le standard JPEG traditionnel repose sur la **Transformée en Cosinus Discrète (DCT, *Discrete Cosine Transform*)**. Malgré les différences structurelles, les deux approches partagent le même principe fondamental : compacter l'énergie de l'image en un nombre réduit de coefficients et éliminer les composantes de moindre importance avec un impact visuel minimal.

L'objectif central de la compression est de réduire le volume de données nécessaire au stockage ou à la transmission d'une image. Ce processus est rendu possible par l'identification et l'élimination des **redondances** structurelles et perceptuelles.

6.7.1 Taxonomie des Redondances

Le développement des algorithmes de compression repose sur l'identification et l'élimination de trois catégories principales de redondance, synthétisées dans la Table 6.6.

Tableau 6.6: Catégories de redondance dans les images numériques et leurs mécanismes d'exploitation respectifs.

Type	Définition	Approche d'exploitation
Spatiale (interpixel)	Forte corrélation et dépendance statistique entre pixels voisins.	DCT, DWT et codage prédictif.
Spectrale (intercanal)	Corrélation statistique entre les canaux de couleur d'une même image.	Transformations d'espace colorimétrique (ex : RGB vers YC_bC_r).
Psychovisuelle	Insensibilité du système visuel humain (SVH) aux variations de haute fréquence et de faible contraste.	Processus de quantification sélective des coefficients.

Selon la préservation de l'information originale après le processus de décodage, les méthodes de compression se divisent en deux classes fondamentales :

- **Sans perte (*lossless*) :** Garantit une reconstruction bit à bit identique à l'image originale. Elle est employée dans des scénarios où l'intégrité des données est strictement critique, comme dans l'imagerie médicale, les diagnostics par imagerie et le stockage de documents textuels.
- **Avec perte (*lossy*) :** Admet l'introduction d'une distorsion contrôlée du signal en échange de taux de compression substantiellement plus élevés. C'est l'approche standard pour les photographies grand public et le *streaming* vidéo, écosystèmes dans lesquels le SVH tolère de légères atténuations haute fréquence sans perception de dégradation de la qualité visuelle.

6.7.2 Transformée en Cosinus Discrète (DCT-II 2D)

La **Transformée en Cosinus Discrète** (DCT) constitue l'opération centrale du standard JPEG. Contrairement à la TFD, qui utilise une base complexe, la DCT repose sur des fonctions trigonométriques purement réelles. Pour un bloc d'image $f(x, y)$ de dimensions $N \times N$, la **DCT-II 2D** mappe le signal spatial vers le domaine des fréquences spatiales, générant la matrice de coefficients $C(u, v)$ au moyen de :

$$C(u,v) = \alpha(u)\,\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y)\, \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right] \tag{6.9}$$

où les facteurs de normalisation orthogonale sont donnés par $\alpha(0) = \sqrt{1/N}$ et $\alpha(k) = \sqrt{2/N}$ pour $k > 0$.

Chaque coefficient $C(u, v)$ quantifie la contribution — ou le « poids » — d'une fréquence spatiale spécifique au sein de ce bloc. Le terme $C(0, 0)$ est appelé **composante DC** et représente l'intensité moyenne du bloc (fréquence nulle). Les autres coefficients, désignés sous le nom de **composantes AC** (*Alternating Current*), correspondent aux fréquences spatiales progressivement plus élevées.

6.7.3 Les fonctions de base de la DCT

D'un point de vue géométrique, la Équation 6.9 réalise la projection du bloc de pixels sur un ensemble de fonctions orthogonales. Pour le cas standard du JPEG ($N = 8$), le bloc spatial est décomposé en une combinaison linéaire de **64 fonctions de base** bidimensionnelles, notées $B_{u,v}(x, y)$ et générées par le produit de fonctions cosinusoïdales :

$$B_{u,v}(x,y) = \cos\left[\frac{\pi(2x+1)u}{16}\right]\cos\left[\frac{\pi(2y+1)v}{16}\right]$$

Ainsi, l'opération inverse peut être interprétée comme la reconstruction exacte du bloc original par la somme pondérée de ces 64 matrices de base, où chaque coefficient $C(u,v)$ agit comme le poids analytique de sa composante harmonique respective.

La **fréquence spatiale** indiquée par les indices (u,v) détermine le nombre de cycles d'oscillation le long des dimensions horizontales et verticales du bloc. Comme illustré dans la Figure 6.23 — dont le code isole chaque base en appliquant la transformation inverse sur des impulsions unitaires —, ces 64 fonctions sont organisées en une matrice 8×8. Le coin supérieur gauche $(u = 0, v = 0)$ présente le motif uniforme de fréquence nulle (DC), tandis que la progression vers la droite (axe u) ou vers le bas (axe v) représente des variations harmoniques progressivement plus importantes, traduisant des transitions rapides, des contours et des textures dans les orientations horizontales, verticales et diagonales.

DCT vs DFT : avantage de la compaction d'énergie

La DCT et la DFT mappent toutes deux un bloc spatial $N \times N$ en une matrice de coefficients de même dimension. Cependant, pour les images naturelles, la DCT présente une plus grande efficacité en matière de **compaction d'énergie** dans les basses fréquences. Cela s'explique par le fait que la DCT suppose implicitement une symétrie paire du signal aux frontières du bloc, ce qui équivaut à une extension périodique continue, minimisant ainsi l'effet de diffusion spectrale (*ringing*). Par conséquent, la plupart des coefficients AC décroissent rapidement vers des valeurs proches de zéro, optimisant le *pipeline* de compression sans introduire de dégradation visuelle perceptible.

```
from scipy.fft import dct, idct  # ligne ajoutée

fig, axes = plt.subplots(8, 8, figsize=(6, 6))
fig.subplots_adjust(hspace=0.05, wspace=0.05)
for i in range(8):
    for j in range(8):
        coef = np.zeros((8, 8)); coef[i, j] = 1
        b = idct(idct(coef.T, norm='ortho').T, norm='ortho')
        axes[i, j].imshow(b, cmap='gray')
        axes[i, j].axis('off')
plt.suptitle("As 64 Bases da DCT 8x8", y=0.92, fontsize=12, fontweight='bold')
plt.show()
```

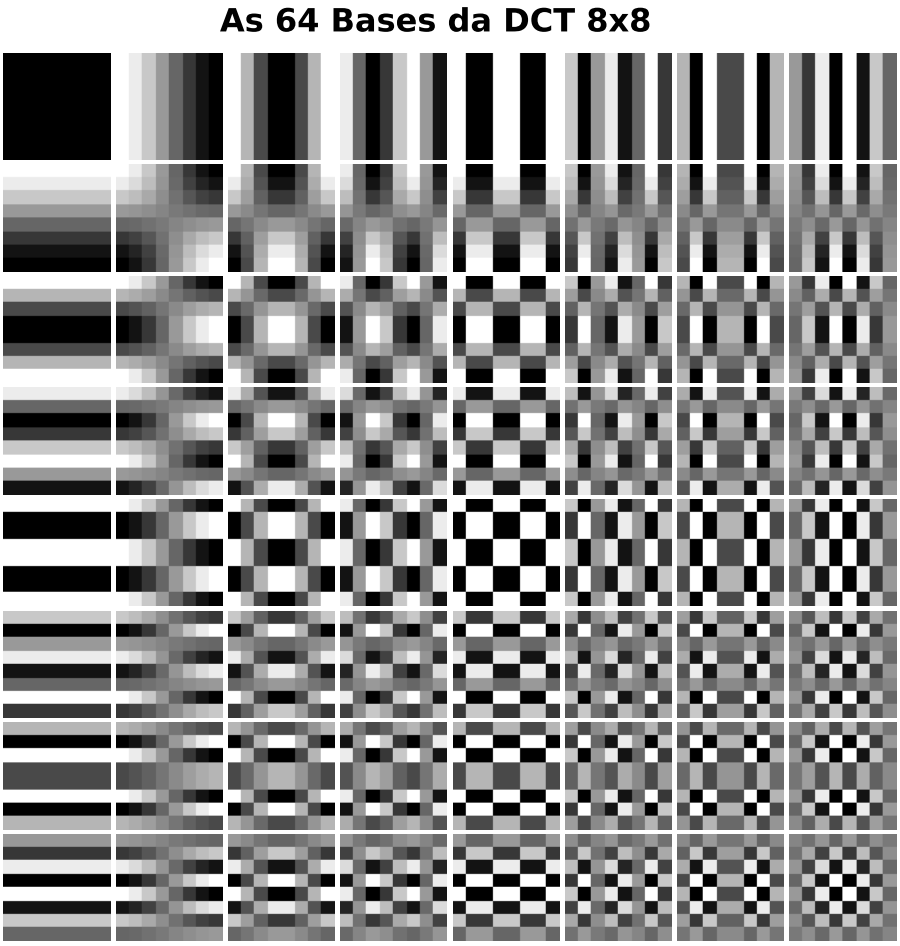


Figure 6.23: L'Alphabet visuel du JPEG : Les 64 fonctions de base de la DCT-II. Le coefficient DC se trouve en haut à gauche (lisse). En descendant et en allant vers la droite, l'oscillation spatiale augmente drastiquement.

6.7.4 Concentration d'Énergie et Reconstruction Progressive

Avant l'application de la DCT, les pixels du bloc d'intensité sont systématiquement translatés (en soustrayant 128 pour les images 8 bits) afin de centrer le signal autour de zéro, éliminant ainsi les composantes continues superflues. Lors du calcul de la DCT sur le bloc résultant, la propriété de **compaction d'énergie** devient évidente : la quasi-totalité de la variance et de l'information de l'image originale se concentre dans le coefficient DC ($C(0,0)$) et dans les premiers harmoniques AC de basse fréquence.

La Figure 6.24 illustre ce phénomène par une reconstruction progressive par troncature abrupte. Au lieu d'utiliser l'ensemble des 64 coefficients, l'algorithme ne conserve que les k premiers composants — sélectionnés sur la base d'un balayage qui privilégie les basses fréquences spatiales — et annule les autres.

La synthèse inverse (**IDCT**) réalisée avec seulement une fraction des coefficients (comme 15 % ou 30 %) est déjà capable de récupérer les structures et l'éclairage macro du bloc de pixels original. À mesure que les harmoniques de fréquences plus élevées sont progressivement réincorporés, les détails fins et les transitions rapides sont restaurés. Ce comportement valide le principe de la compression perceptuelle : les hautes fréquences écartées possèdent peu d'énergie et leur absence, en conditions normales, génère un impact visuel secondaire sur la perception de l'observateur.

```
from scipy.fft import dct, idct

def dct2(bloco):
    """DCT-II 2D ortogonal (separável)."""
```

```python
    return dct(dct(bloco.T, norm='ortho').T, norm='ortho')

def idct2(coefs):
    """IDCT-II 2D ortogonal."""
    return idct(idct(coefs.T, norm='ortho').T, norm='ortho')

#   Bloco 8×8 centralizado da imagem
cy, cx = img_gray.shape[0]//2, img_gray.shape[1]//2
bloco  = img_gray[cy:cy+8, cx:cx+8].astype(np.float64) - 128.0

C = dct2(bloco)

print("Coeficientes DCT do bloco 8×8:")
print(np.round(C).astype(int))
print(f"\nEnergia DC     : {C[0,0]**2:.1f}")
print(f"Energia total  : {(C**2).sum():.1f}")
print(f"Fração no DC   : {C[0,0]**2 / (C**2).sum():.1%} ← concentração de energia")

#   Reconstrução progressiva
imgs_rec  = [cv2.normalize((bloco+128).astype(np.uint8), None, 0, 255, cv2.NORM_MINMAX)]
titles_rec = ["Bloco original\n(8×8 pixels)"]

for keep in [1, 4, 10, 20, 40, 64]:
    C_trunc = np.zeros_like(C)
    indices = sorted([(u,v) for u in range(8) for v in range(8)], key=lambda p: p[0]+p[1])
    for u, v in indices[:keep]:
        C_trunc[u, v] = C[u, v]
    rec = np.clip(idct2(C_trunc) + 128, 0, 255).astype(np.uint8)
    imgs_rec.append(rec)
    titles_rec.append(f"{keep} coef.\n({keep/64:.0%} do total)")

mm.show(imgs_rec, titles=titles_rec, cols=4, figsize=(12, 7))
```

```
Coeficientes DCT do bloco 8×8:
[[ 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]]

Energia DC     : 202725.1
Energia total  : 639868.0
Fração no DC   : 31.7% ← concentração de energia
```

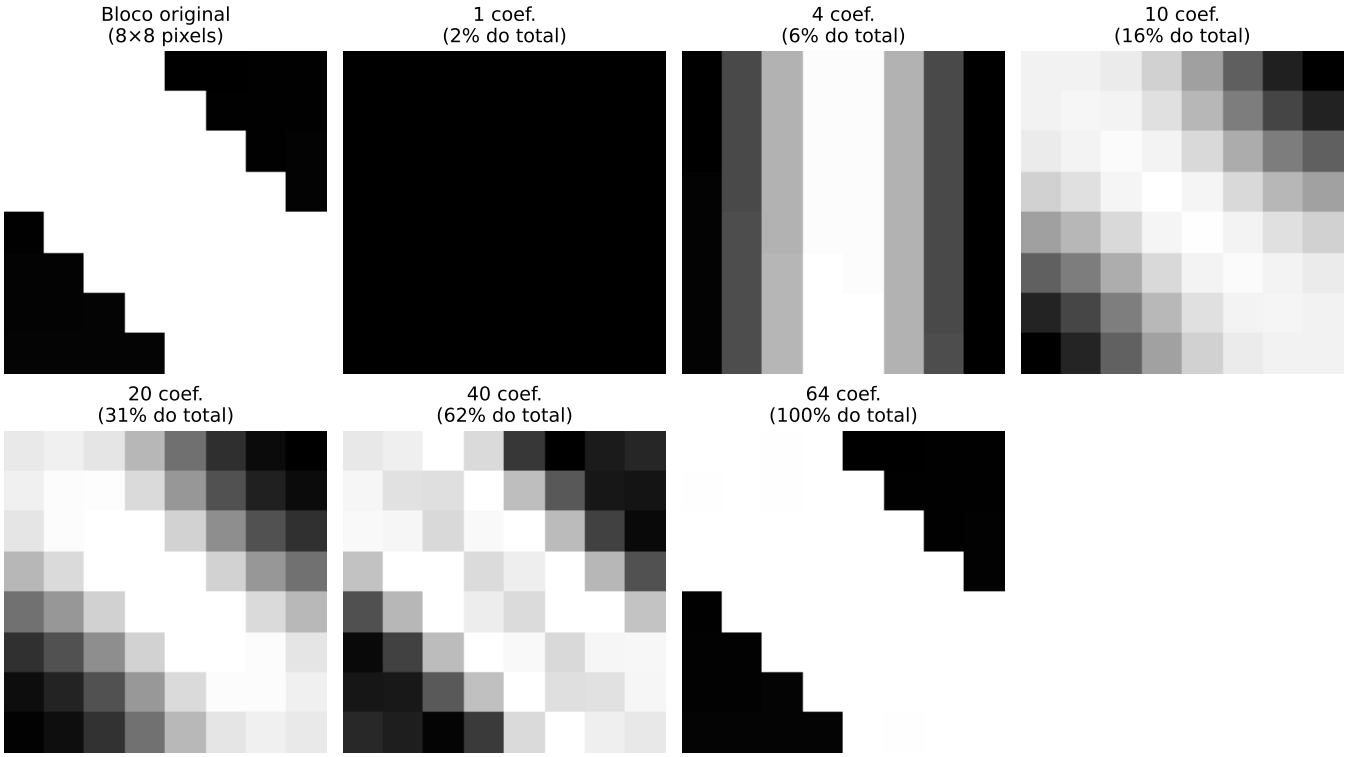


Figure 6.24: DCT 2D em bloco 8×8: coeficientes e reconstrução progressiva.

6.7.5 Le *Pipeline* de Compression JPEG

La norme JPEG opère en divisant l'image en blocs disjoints de 8×8 pixels, traités par une séquence de transformations spatiales, perceptuelles et statistiques. Le *pipeline* complet de codage est structuré en six étapes principales :

$$\text{RGB} \xrightarrow{(1)\ YC_bC_r} \xrightarrow{(2)\ \text{Sous-échantillonnage}} \xrightarrow{(3)\ \text{Blocs } 8\times8} \xrightarrow{(4)\ \text{DCT}} \xrightarrow{(5)\ \text{Quantification}} \xrightarrow{(6)\ \text{Codage entropique}}$$

La Table 6.7 détaille la fonction analytique et le fondement perceptuel qui justifient chacune de ces étapes.

Tableau 6.7: Étapes du *pipeline* de compression JPEG et leurs fondements de conception respectifs.

Étape	Opération	Fondement perceptuel et statistique
1	Conversion $RGB \to YC_bC_r$	Sépare la luminance (Y) de la chrominance (C_b, C_r). Le système visuel humain (SVH) présente une plus grande sensibilité aux variations de luminosité qu'aux variations de couleur.
2	Sous-échantillonnage de la chrominance (ex. : 4:2:0)	Réduit la résolution spatiale des canaux de couleur de moitié, en éliminant des données redondantes avec un impact visuel négligeable.

Étape	Opération	Fondement perceptuel et statistique
3–4	Centrage et application de la DCT 8×8	Translates les pixels dans l'intervalle $[-128, 127]$ et compresse l'énergie spectrale du bloc dans les coefficients de basse fréquence.
5	Quantification linéaire sélective	Divise chaque coefficient $C(u, v)$ par l'élément correspondant de la matrice $Q(u, v)$, avec arrondi entier. Constitue la principale source de compression avec perte.
6	Balayage en zigzag et codage	Ordonne les coefficients quantifiés pour maximiser les séquences nulles consécutives, optimisant le codage par longueur de plage (RLE) et le codage de Huffman.

La **matrice de quantification** $Q(u, v)$ est le mécanisme central de contrôle du compromis entre taux de compression et qualité visuelle. Dans l'algorithme pratique de la Figure 6.25, le facteur de qualité spécifié par l'utilisateur (échelle de 1 à 100) est converti en un scalaire qui paramètre la sévérité de la matrice Q. Des valeurs de qualité réduites élargissent les diviseurs de $Q(u, v)$, forçant le tronquement en masse des coefficients AC à zéro. Lorsque cette élimination est excessive, la discontinuité aux frontières des blocs adjacents n'est pas atténuée lors de la reconstruction, générant ce que l'on appelle les **artefacts de bloc** (*blocking artifacts*).

La Logique du Balayage en Zigzag

L'efficacité du codeur entropique subséquent à la quantification dépend directement de l'ordonnancement des données. Comme la DCT concentre l'énergie vitale dans le coin supérieur gauche de la matrice (basses fréquences) et repousse les coefficients nuls vers les extrémités opposées, la lecture linéaire par lignes ou par colonnes fragmenterait les séquences de zéros.

L'ordonnancement en zigzag résout cette limitation en parcourant la matrice en diagonale, dans l'ordre croissant de la fréquence spatiale. Ce mappage regroupe les coefficients significatifs au début du vecteur et concentre les coefficients nuls en une seule séquence continue à la fin du tableau, permettant à l'algorithme RLE de coder de grands blocs de données de manière compacte et efficace.

ℹ Qu'est-ce que le RLE ?

RLE (*Run-Length Encoding*) est une technique de compression sans perte qui code des séquences consécutives de valeurs identiques — en particulier des **zéros** — comme une paire (compteur, valeur). Dans le JPEG, après le balayage en zigzag, les coefficients quantifiés sont organisés de sorte que les zéros se concentrent à la fin du vecteur. Le RLE compresse ensuite cette longue course de zéros avec une extrême efficacité, optimisant le stockage et la transmission de l'image compressée.

```
import numpy as np
import cv2
from scipy.fft import dct, idct

#   Chargement Sécurisé de l'Image de la Caméra (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()

# Redimensionne légèrement en 256x256 pour maintenir le standard et la vitesse des tests
précédents
img_gray = cv2.resize(img_gray, (256, 256))

#   Table de quantification de luminance (standard JPEG)
Q_luma = np.array([
    [16,11,10,16,24,40,51,61],
    [12,12,14,19,26,58,60,55],
    [14,13,16,24,40,57,69,56],
    [14,17,22,29,51,87,80,62],
    [18,22,37,56,68,109,103,77],
    [24,35,55,64,81,104,113,92],
    [49,64,78,87,103,121,120,101],
    [72,92,95,98,112,100,103,99]
], dtype=np.float64)

def dct2(bloco):
    """DCT-II 2D orthogonal (séparable)."""
    return dct(dct(bloco.T, norm='ortho').T, norm='ortho')

def idct2(coefs):
    """IDCT-II 2D orthogonal."""
    return idct(idct(coefs.T, norm='ortho').T, norm='ortho')

def jpeg_compress_block(bloco, Q_table):
    """DCT → quantification → déquantification → IDCT en bloc 8×8."""
    C  = dct2(bloco.astype(np.float64) - 128)
    Cq = np.round(C / Q_table) * Q_table    # quantifie et déquantifie
    return np.clip(idct2(Cq) + 128, 0, 255)

def jpeg_quality_compress(img, qualidade=50):
    """JPEG simplifié : compresse l'image entière par blocs 8×8."""
    if qualidade < 50:
        escala = 5000 / qualidade
    else:
        escala = 200 - 2 * qualidade
    # Corrigé de 'scala' à '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)

#   Comparaison des facteurs de qualité
qualidades = [10, 25, 50, 75, 90]
imgs_jpeg  = [img_gray]
titles_jpeg = ["Original\n(Cameraman)"]

for q in qualidades:
    rec  = jpeg_quality_compress(img_gray, qualidade=q)
    psnr = cv2.PSNR(img_gray, rec)
```

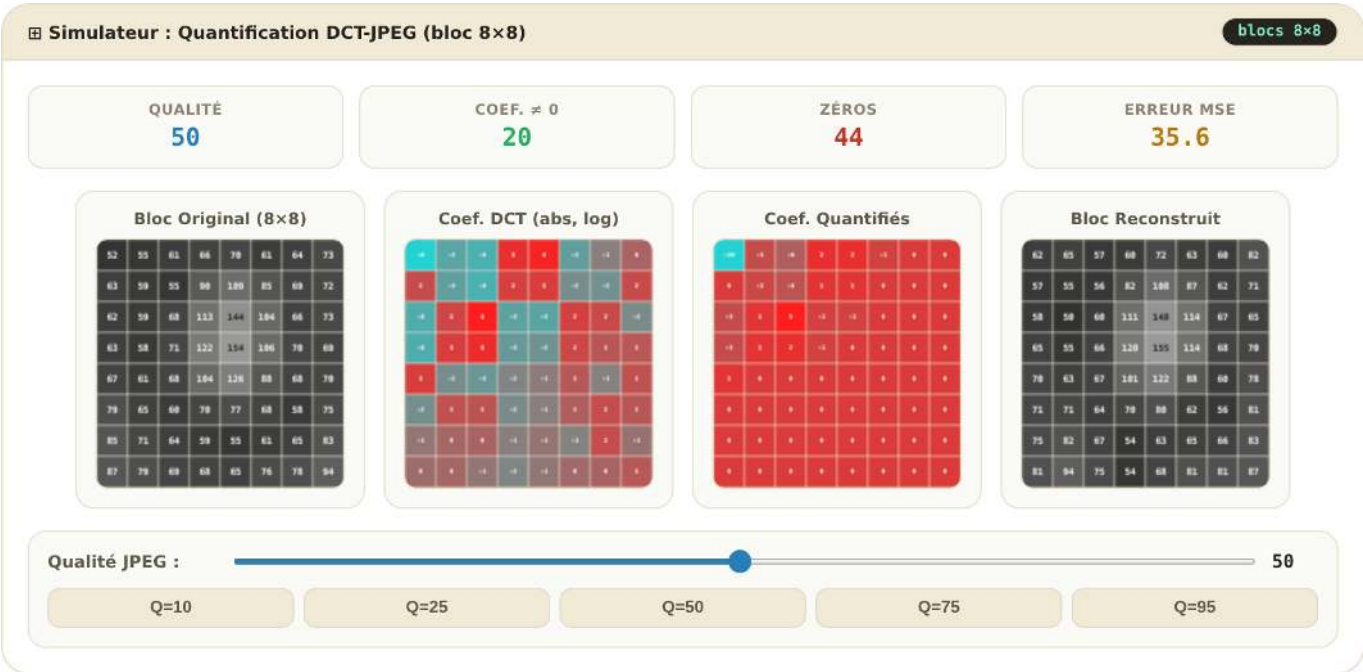
```
    imgs_jpeg.append(rec)
    titles_jpeg.append(f"Q={q}\nPSNR={psnr:.1f}dB")

mm.show(imgs_jpeg, titles=titles_jpeg, cols=3, figsize=(14, 10))
```

Figure 6.25: *Pipeline* JPEG simplifié appliqué à l'image classique du *Cameraman* : DCT en blocs 8×8, quantification avec différents facteurs de qualité et reconstruction via IDCT. Les artefacts de bloc (*blocking artifacts*) deviennent visuellement évidents avec des facteurs de qualité réduits ($Q = 10$ et $Q = 25$).

6.7.6 Simulateur interactif : Quantification DCT

Le simulateur de la Figure 6.26 permet d'explorer l'impact du processus de quantification sur un bloc 8×8 extrait d'une image réelle, en synthétisant en temps réel les composantes suivantes :

- **Bloc original et reconstruit :** Représentation directe des pixels dans le domaine spatial en niveaux de gris [0, 255].
- **Coefficients DCT :** Distribution de l'énergie mappée de manière logarithmique sur un dégradé chromatique, mettant en évidence la concentration de l'intensité dans le coin supérieur gauche (basses fréquences).
- **Coefficients quantifiés :** Affichage des valeurs entières résultant de la division par la matrice $Q(u, v)$, rendant visuellement explicite l'apparition massive de coefficients nuls (en tons sombres) à mesure que le facteur de qualité est réduit.
- **Métriques de compression :** Panneau de surveillance qui quantifie l'Erreur Quadratique Moyenne (MSE), le nombre de coefficients préservés et le volume de zéros générés pour le codage entropique.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap05/sim-05-dct.html

Figure 6.26: Simulateur interactif de compression DCT-JPEG : ajustez le facteur de qualité et visualisez en temps réel les coefficients nuls, le bloc reconstruit et l'erreur de quantification.

6.8 Comparação de Formatos de Imagem

Le choix d'un format de stockage numérique impacte directement le compromis entre qualité visuelle, taille de fichier et coût computationnel de décodage. Les trois formats les plus pertinents pour les architectures *web* et les systèmes de calcul visuel sont le JPEG, le PNG et le WebP.

6.8.1 Caractéristiques des Formats

La Table 6.8 synthétise les propriétés structurelles des principaux formats d'image matriciels.

Tableau 6.8: Comparaison structurelle entre les principaux formats d'image matriciels.

Caractéristique	JPEG	PNG	WebP
Compression	Avec perte	Sans perte	Avec et sans perte.
Transparence (canal alpha)	Non	Oui	Oui.
Prise en charge de l'animation	Non	Limitée (APNG)	Oui.
Algorithme de base	DCT + Huffman	DEFLATE (LZ77 + Huffman)	VP8 / VP8L.
Idéal pour	Photographie	Graphiques, texte et icônes	Usage universel en environnement Web.
Moins adapté pour	Texte et contours nets	Images photographiques complexes	Compatibilité héritée.

6.8.2 Métriques d'Évaluation de la Qualité

Deux métriques objectives sont largement adoptées pour quantifier la distorsion introduite par les processus de compression :

Pic du Rapport Signal-à-Bruit (PSNR, *Peak Signal-to-Noise Ratio*) :

$$\text{PSNR} = 10\,\log_{10}\left(\frac{L^2}{\text{MSE}}\right) \quad [\text{dB}] \tag{6.10}$$

où $L = 255$ pour les images quantifiées sur 8 bits et MSE représente l'**Erreur Quadratique Moyenne** (*Mean Squared Error*). Des valeurs de PSNR supérieures à 40 dB indiquent une excellente fidélité ; entre 30 dB et 40 dB, elles représentent une bonne qualité ; et des valeurs inférieures à 30 dB correspondent à des dégradations visuelles facilement perceptibles.

Indice de Similarité Structurale (SSIM, *Structural Similarity Index*) :

$$\text{SSIM}(f, g) = \frac{(2\mu_f\mu_g + c_1)(2\sigma_{fg} + c_2)}{(\mu_f^2 + \mu_g^2 + c_1)(\sigma_f^2 + \sigma_g^2 + c_2)} \tag{6.11}$$

Le SSIM évalue des fenêtres locales de l'image sur la base de trois composantes complémentaires : la **luminance** (μ_f, μ_g), le **contraste** (σ_f, σ_g) et la **structure** (σ_{fg}), pondérées par des constantes de stabilité c_1 et c_2. L'indice varie dans l'intervalle $[-1, 1]$, où l'unité représente l'identité parfaite. Contrairement au PSNR, le SSIM prend en compte l'organisation spatiale des erreurs, s'alignant ainsi sur la perception du système visuel humain (SVH).

> **PSNR vs SSIM : Application de Métriques Perceptuelles**
>
> Le PSNR possède une formulation mathématique simple et un faible coût computationnel ; toutefois, il tend à surestimer la qualité des images présentant des distorsions localisées ou à la sous-estimer en cas de variations globales de luminosité tolérées par l'observateur. Le SSIM modélise plus fidèlement la perception biologique, mais exige un effort de traitement plus important. Pour des analyses rigoureuses des codecs, il est recommandé de rapporter les deux métriques statistiques à titre complémentaire.

6.8.3 Inspection Visuelle : Nature des Artefacts de Compression

La nature mathématique du codec détermine le type de dégradation introduit à des débits binaires réduits. Comme illustré dans la Figure 6.27, la compression agressive via DCT dans le standard JPEG segmente l'image en mailles rigides, générant les **artefacts de bloc** (*blocking artifacts*). En revanche, les algorithmes basés sur la codification prédictive ou les représentations soumises à des transformées spatiales avancées (comme le WebP et le JPEG 2000) éliminent les discontinuités de bloc, mais introduisent une perte de texture fine et des flous caractéristiques autour des contours à fort contraste.

```
import os
import cv2

# Garantit l'existence du répertoire et enregistre les fichiers compressés
os.makedirs("imagens/comp_test", exist_ok=True)
cv2.imwrite("imagens/comp_test/camera_q10.jpg", img_gray, [cv2.IMWRITE_JPEG_QUALITY, 10])
cv2.imwrite("imagens/comp_test/camera_q10.webp", img_gray, [cv2.IMWRITE_WEBP_QUALITY, 10])

# Extraction de la région d'intérêt pour visualisation des artefacts (Zoom 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 (Artefact de Bloc)", "WebP Q=10 (Adoucissement)"],

        cols=3, figsize=(14, 5))
```

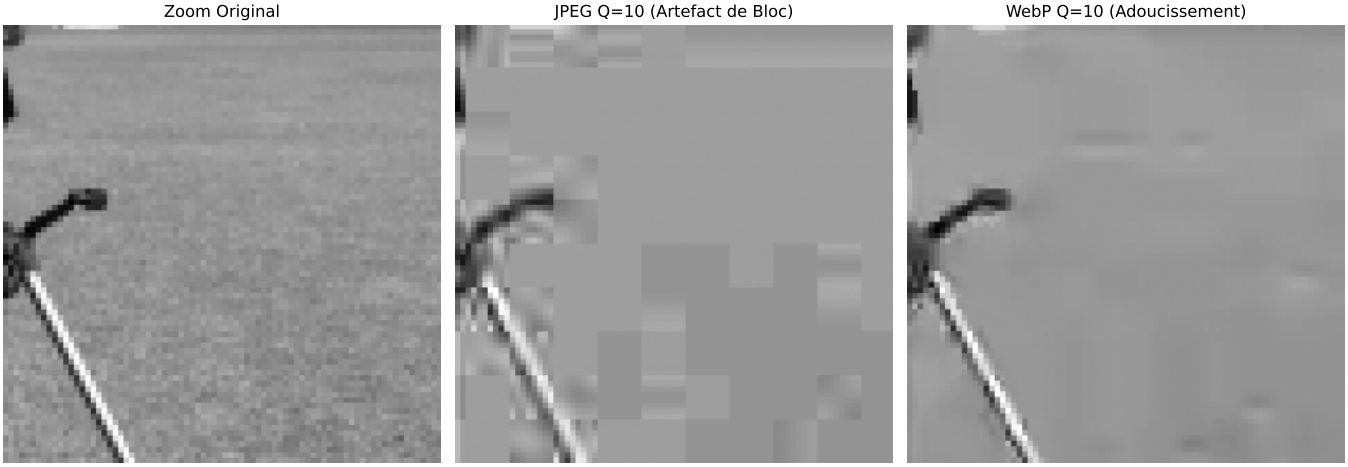


Figure 6.27: Analyse comparative des artefacts de compression sous facteur de qualité réduit ($Q = 10$). À gauche, on observe l'artefact de bloc caractéristique de la discrétisation par DCT dans le JPEG. À droite, on met en évidence l'effet d'atténuation et d'adoucissement des bords intrinsèque au format WebP.

6.8.4 Évaluation Quantitative et Spatiale de la Compression

La validation des algorithmes de compression avec perte exige une analyse qui corrèle le coût de stockage à la fidélité du signal reconstruit. Cette évaluation est réalisée de manière complémentaire à travers des courbes de performance globale et par la cartographie locale des distorsions induites par les codeurs.

6.8.4.1 Courbes Débit-Distorsion

La Figure 6.28 présente l'évaluation empirique du *pipeline* JPEG et WebP au moyen de **courbes débit-distorsion**, qui surveillent le gain de compression (taille du fichier en Ko) en fonction du PSNR. Le format PNG sert de ligne de base idéale (PSNR $= \infty$), car sa nature *lossless* empêche toute dégradation, bien qu'il exige un volume de données substantiellement plus important.

L'analyse des courbes démontre la supériorité et l'efficacité du standard WebP par rapport au JPEG traditionnel : pour atteindre un même niveau de fidélité mathématique (comme la plage d'excellente qualité, où PSNR > 40 dB), le codeur WebP génère des fichiers significativement plus petits. Ce comportement traduit l'impact pratique de l'évolution des algorithmes sur l'optimisation des systèmes de transmission et de stockage numérique.

```
import os
import cv2
import matplotlib.pyplot as plt

# Garantit l'existence du répertoire de tests
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})

#   Génération de la courbe taux-distorsion
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"\nTaille brute (sans compression) : {img_gray.nbytes/1024:.0f} Ko")
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}")
```

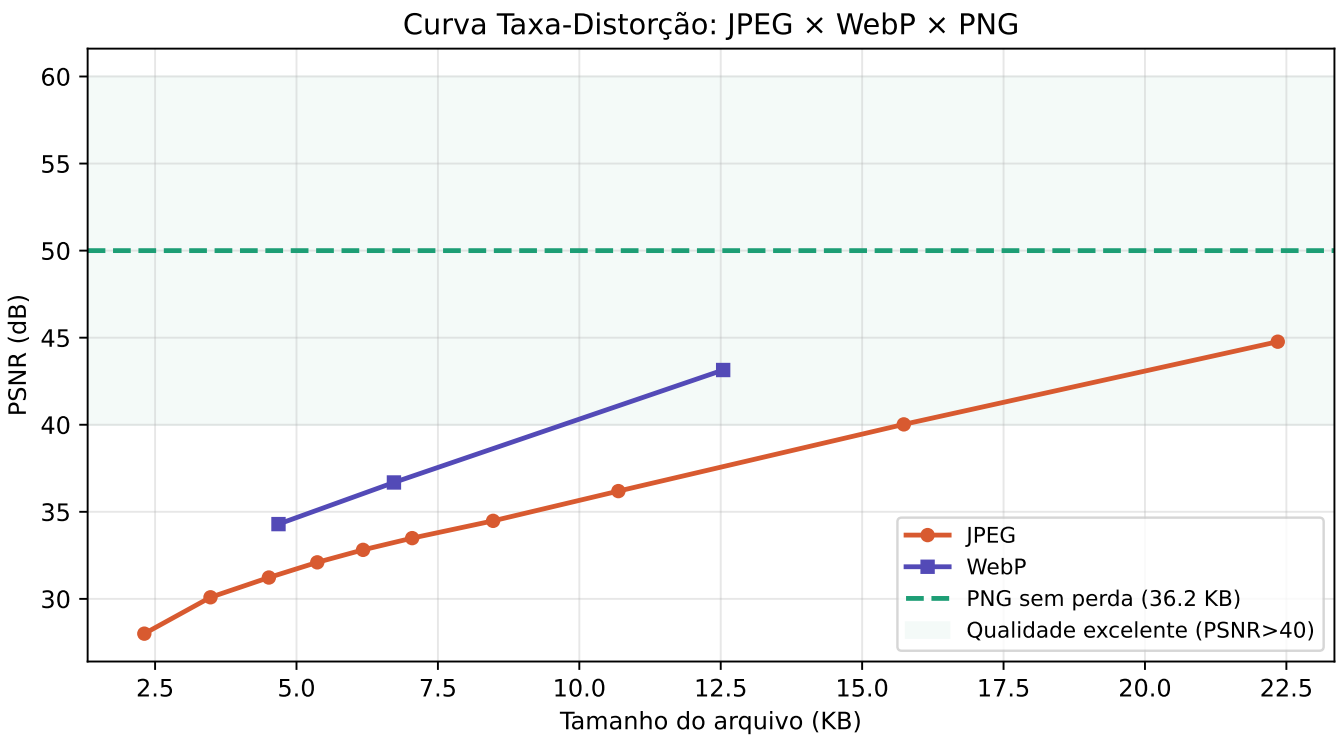


Figure 6.28: Courbe taux-distorsion : PSNR vs taille de fichier pour JPEG, WebP et PNG appliquée à l'image du *Cameraman*.

```
Taille brute (sans compression) : 64 Ko

 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
```

Taille originale de l'image

L'image *Cameraman* (256 × 256 pixels en niveaux de gris) occupe **64 Ko** en format brut (sans compression). À titre de référence, le PNG *lossless* compresse ce volume à **36,2 Ko** — mettant en évidence que la compression sans perte réduit déjà significativement le stockage pour les images comportant des régions homogènes. En revanche, les formats avec perte (JPEG et WebP) atteignent des tailles encore plus réduites : le JPEG avec une qualité de 95 occupe 22,3 Ko (PSNR 45 dB), tandis que le WebP avec une qualité de 90 atteint 12,5 Ko avec un PSNR équivalent, démontrant sa supériorité en matière d'efficacité de compression.

6.8.4.2 Cartographie Spatiale des Erreurs et Corrélation Perceptuelle

Bien que le PSNR offre un indicateur numérique rapide, les métriques globales ne parviennent pas à distinguer comment la perte d'informations se répartit géométriquement sur l'image. La Figure 6.29 résout cette limitation en associant les reconstructions à différentes qualités à leurs cartes d'erreur absolue respectives et au SSIM.

Les cartes résiduelles — obtenues par la différence absolue normalisée entre l'image originale et l'image compressée — révèlent la signature spatiale intrinsèque de chaque architecture de codage :

- **À des qualités élevées ($Q = 95$ à $Q = 75$) :** Les distorsions se concentrent principalement autour des transitions abruptes d'intensité (bords), résultant du repliement spectral dû à l'élimination des hautes fréquences. L'indice SSIM reste proche de l'unité, attestant de l'intégrité des structures originales.
- **À des qualités agressives ($Q = 50$ à $Q = 25$) :** L'erreur adopte une structure de maillage orthogonal régularisé. Ce motif géométrique met en évidence l'apparition des **artefacts de bloc** (*blocking artifacts*), indiquant que la quantification sévère a corrompu la corrélation spatiale entre les blocs adjacents de 8×8 pixels.

Le SSIM capture cette dégradation morphologique de manière beaucoup plus sensible que le PSNR, pénalisant le score final à mesure que l'organisation structurelle et les textures fines — auxquelles le système visuel humain est hautement réactif — sont éliminées par le codeur.

```
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

# Garantit l'existence du répertoire de test
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"

    # ENREGISTREMENT FORCÉ : Génère et enregistre le JPEG avec la qualité actuelle au bon
    chemin
    img_compactada = jpeg_quality_compress(img_gray, qualidade=q)
    cv2.imwrite(path, img_compactada)

    # Lecture sécurisée du fichier nouvellement enregistré
    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)
```

```
    # Différence absolue normalisée pour mettre en évidence la structure spatiale de l'erreur
    diff_vis = cv2.normalize(np.abs(img_gray.astype(float) - rec.astype(float)),
                             None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

    imgs_ssim  += [rec, diff_vis]
    titles_ssim += [f"Q={q}\nPSNR={psnr_v:.1f}dB | SSIM={ssim_v:.3f}",
                    f"Mapa de erro (Q={q})\n(Bordas e blocagem)"]

mm.show(imgs_ssim, titles=titles_ssim, cols=3, figsize=(14, 14))
```

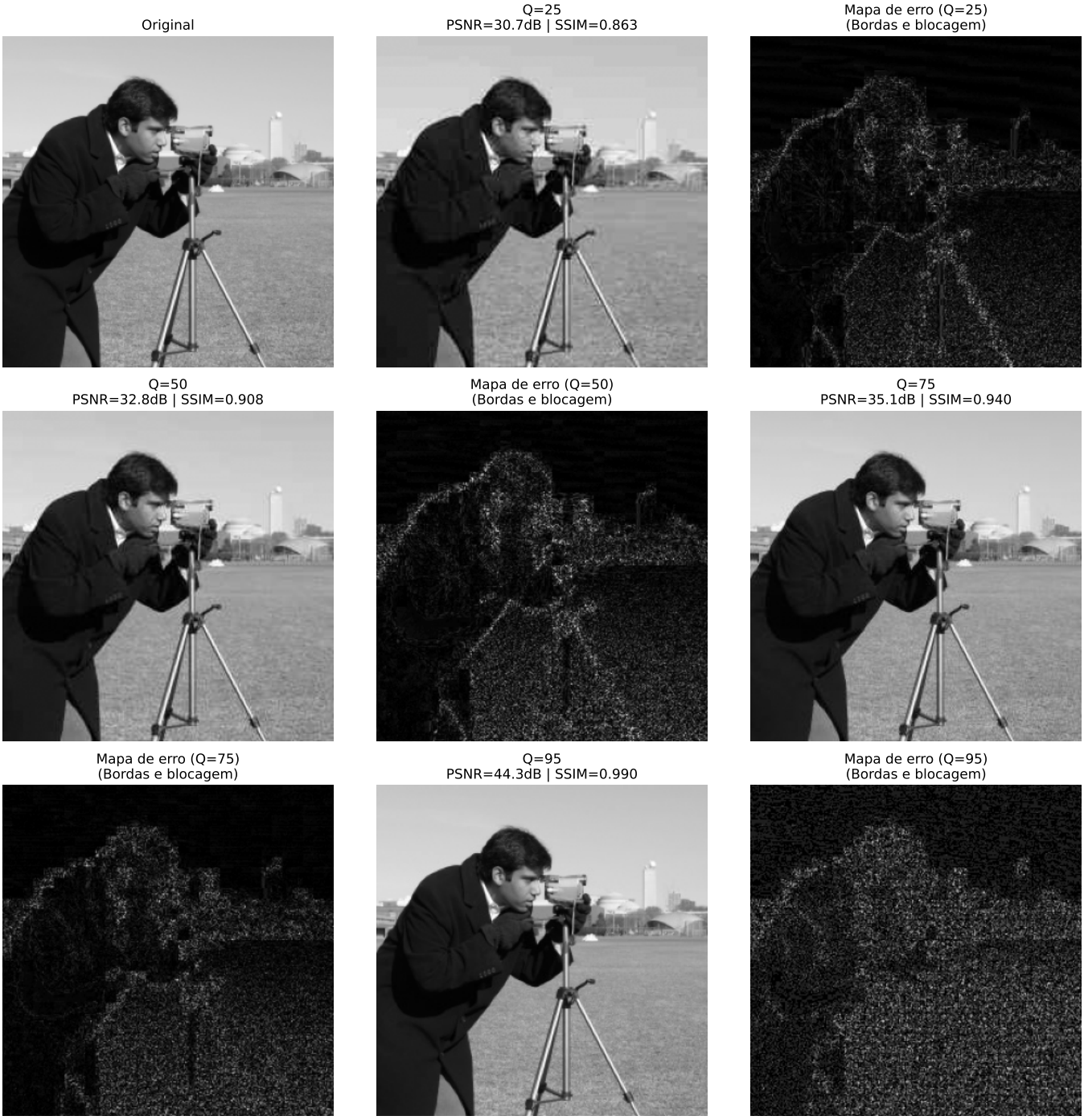


Figure 6.29: Analyse spatiale de la dégradation : images reconstruites et cartes d'erreur absolue normalisées associées pour différents facteurs de qualité JPEG.

Interprétation des cartes d'erreur

Les cartes d'erreur présentées ont été **normalisées individuellement** (`cv2.NORM_MINMAX`) afin de maximiser le contraste visuel et de révéler la structure spatiale des distorsions. Cela signifie que :

- Pour **Q=95**, l'erreur absolue est de l'ordre de **0,5 à 1,5 niveaux de gris** (imperceptible visuellement), mais la normalisation l'amplifie en noir et blanc pour mettre en évidence sa localisation au niveau des bords et des transitions.
- Pour **Q=25**, l'erreur absolue est **10 à 20 fois plus grande** (5 à 15 niveaux de gris), mais la normalisation l'amène également dans la même plage [0, 255].

Par conséquent, **l'intensité du blanc dans les cartes n'est PAS comparable entre différentes qualités** — les cartes servent uniquement à révéler la **signature spatiale** de l'erreur (bords vs blocs), et non son ampleur. L'ampleur correcte est fournie par les valeurs PSNR et SSIM, qui montrent clairement que Q=95 présente une erreur bien inférieure à celle de Q=25.

Synthèse — Compression JPEG

Le processus de compression dans le standard JPEG repose sur l'application combinée de transformations spatiales, perceptuelles et statistiques pour réduire les redondances d'une image. La Table 6.9 résume le rôle de chaque étape dans le *pipeline* et son impact respectif sur la réduction des données.

Tableau 6.9: Synthèse des étapes du *pipeline* de compression JPEG et de leurs impacts respectifs.

Étape	Opération Analytique	Mécanisme de Gain / Compression
Conversion YC_bC_r	Isolation des canaux de luminance et de chrominance.	Modélise la perception du système visuel humain (SVH), permettant de traiter la couleur et la luminosité de manière indépendante.
Sous-échantillonnage 4:2:0	Réduction de la résolution spatiale des canaux de couleur (C_b et C_r).	Élimine environ 50 % des données brutes avec un impact visuel minimal.
DCT 8×8	Mappage du domaine spatial vers le domaine des fréquences spatiales.	Compactage de l'énergie, concentrant l'information essentielle dans les premiers coefficients.
Quantification Linéaire	Division entière des coefficients par une matrice de pondération $Q(u, v)$.	Principale source de compression avec perte ; élimine les hautes fréquences imperceptibles.
Codage Entropique	Application d'algorithmes RLE et de codage de Huffman.	Compression statistique sans perte, optimisée par les longues séquences de coefficients nuls.

Artefacts de dégradation caractéristiques

L'application de taux de compression excessivement agressifs (facteurs de qualité réduits) introduit des distorsions prévisibles dans l'image reconstruite, découlant des limitations mathématiques du modèle :

- **Artefacts de bloc (*blocking artifacts*) :** Discontinuités géométriques visibles aux frontières des blocs de 8×8 pixels, causées par la perte de corrélation spatiale après la quantification sévère des composantes AC.
- **Effet de sonnerie (*ringing*) :** Oscillations fantômes ou distorsions de « fumée » autour des bords nets et à fort contraste, provoquées par l'élimination brutale des harmoniques de haute fréquence nécessaires pour reconstruire les fonctions échelon.

- **Perte de texture fine :** Atténuation des détails à haute fréquence et à faible contraste (comme les pelouses, les tissus ou la porosité), donnant aux régions initialement texturées un aspect excessivement lisse ou homogénéisé.

6.9 Application Pratique : Débruitage par Filtrage Hybride

En réunissant les techniques consolidées tout au long de ce chapitre, on présente un *pipeline* complet de **restauration d'images** qui combine l'analyse spectrale dans le domaine fréquentiel avec le filtrage adaptatif dans le domaine spatial. L'objectif est d'atténuer un bruit mixte (composé d'une dégradation gaussienne et d'une interférence périodique) tout en préservant au maximum les détails structurels de l'image originale.

$$\text{Image Bruitée} \xrightarrow{\text{FFT2}} \xrightarrow{\text{Filtre Notch Gaussien}} \xrightarrow{\text{IFFT2}} \xrightarrow{\text{Filtre Bilatéral}} \text{Image Restaurée}$$

Évaluation Complémentaire : PSNR vs. SSIM

La paire de métriques statistiques PSNR et SSIM fournit une évaluation qualitative et morphologique complémentaire du processus de restauration :

- **PSNR :** Pénalise uniformément l'écart quadratique moyen pixel par pixel.
- **SSIM :** Évalue la préservation des structures locales perceptuellement pertinentes (luminance, contraste et contours).

En pratique, il existe un compromis analytique (*trade-off*) entre **réduction du bruit** et **préservation des détails** : des filtres spatiaux excessivement agressifs atténuent bien le bruit haute fréquence, mais dégradent les textures fines et lissent les contours nets — ce qui **réduit simultanément** à la fois le PSNR et le SSIM par rapport à l'image originale. Le défi de la conception de filtres est de trouver le point d'équilibre qui maximise les deux métriques, garantissant une restauration fidèle et visuellement agréable.

6.9.1 Analyse des performances et conclusion du chapitre

Les résultats numériques et visuels générés par Figure 6.30 démontrent la pertinence pratique d'associer différents domaines de traitement. L'insertion simultanée de bruit périodique et stochastique corrompt les propriétés morphologiques du signal, réduisant sévèrement les indices de similarité et le rapport signal-bruit de l'image de référence.

L'isolation et la suppression des pics harmoniques dans le domaine fréquentiel au moyen du masque *notch* éliminent les franges d'interférence sinusoïdales réparties sur l'espace bidimensionnel. Comme le montrent les données imprimées de Figure 6.30, ce filtrage chirurgical induit un saut immédiat et substantiel de la métrique PSNR. Toutefois, le bruit gaussien haute fréquence reste actif de manière homogène dans le spectre, exigeant une approche complémentaire.

La restauration finale est consolidée dans le domaine spatial avec l'introduction du filtre bilatéral. Contrairement aux opérateurs passe-bas classiques (tels que le filtre gaussien ou le filtre moyenneur), qui lisseraient indifféremment le bruit et les contours structurels, le filtrage bilatéral calcule des poids pondérés par la proximité géométrique et par la différence d'intensité radiométrique. Ce comportement adaptatif atténue les fluctuations stochastiques résiduelles dans les régions de transition douce et préserve la netteté des bords spatiaux.

La convergence des deux approches aboutit à une **amélioration substantielle et simultanée** du PSNR et du SSIM par rapport à l'image bruitée — bien que les valeurs finales demeurent inférieures à celles de l'image originale (PSNR $= \infty$, SSIM $= 1{,}0$), en raison de la perte inévitable d'informations spectrales et texturales lors des processus de filtrage. L'atténuation douce (gaussienne) des pics dans le spectre évite les artefacts de *ringing*, tandis que le filtre bilatéral élimine le bruit stochastique résiduel sans compromettre la netteté des bords. Les résultats prouvent l'efficacité et la complémentarité pratique des outils d'analyse fréquentielle

présentés dans ce chapitre, démontrant que le filtrage hybride (fréquence + spatial) est supérieur à toute approche isolée pour la restauration d'images dégradées par un bruit mixte.

```
import numpy as np
import cv2

try:
    from skimage.metrics import structural_similarity as ssim_sk
except ImportError:
    import subprocess
    subprocess.run(["pip", "install", "scikit-image", "-q"])
    from skimage.metrics import structural_similarity as ssim_sk

def suprimir_pico_gaussiano(mask, cy, cx, sigma=3.0):
    yy, xx = np.ogrid[:mask.shape[0], :mask.shape[1]]
    dist = np.sqrt((yy - cy)**2 + (xx - cx)**2)
    notch = np.exp(-dist**2 / (2 * sigma**2))
    mask *= (1 - notch)
    return mask

#   1. Construction du bruit mixte
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. Spectre et identification des pics
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 gaussien sur les pics périodiques
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. Filtre bilatéral : suppression du bruit gaussien résiduel
img_den = cv2.bilateralFilter(img_notch, d=7, sigmaColor=25, sigmaSpace=7)

#   Calcul des métriques de validation
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"Bruitée\nPSNR={psnr_n:.1f} dB",
        "Spectre\n(pics visibles)",
        "Masque notch\n(gaussien doux)",
        f"Après notch\nPSNR={psnr_no:.1f} dB",
        f"Notch + bilatéral\nPSNR={psnr_d:.1f} dB  SSIM={ssim_d:.3f}"
    ],
    cols=6, figsize=(20, 4)
)
```

```
                Etapa | PSNR (dB) |   SSIM
-----------------------------------------
 Ruidosa (gauss+per) |     20.19 | 0.3486
          Após notch |     24.47 | 0.4715
   Notch + bilateral |     28.93 | 0.7426
```

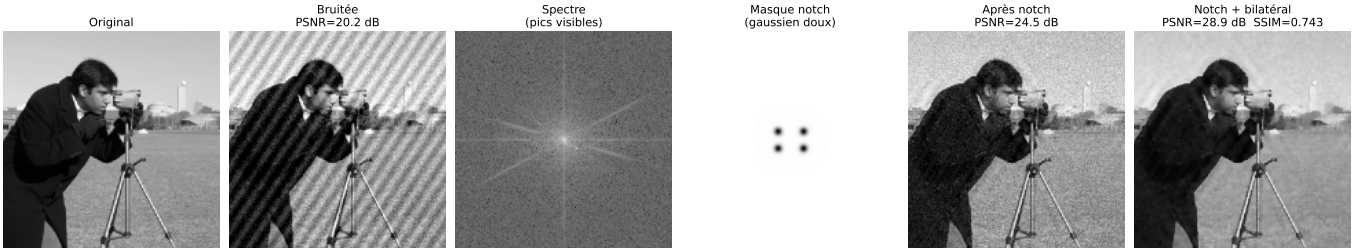


Figure 6.30: *Pipeline* complet de suppression du bruit mixte : (1) ajout de bruit gaussien et périodique ; (2) identification des pics d'interférence dans le spectre de fréquences ; (3) application d'un masque *notch* avec atténuation gaussienne douce ; (4) post-traitement via filtre bilatéral pour élimination du bruit stochastique résiduel.

6.10 Résumé du Chapitre

La transition du **domaine spatial** vers le **domaine fréquentiel** révèle la distribution spectrale de l'énergie de l'image, établissant ainsi la base analytique pour le filtrage avancé, la restauration et la compression des données. L'articulation structurelle de ces concepts est synthétisée dans la carte conceptuelle de la Figure 6.31.

Fundamentos Essenciais

- **TFD et Perception Visuelle :** Le spectre décompose l'image en composantes harmoniques. La **phase** conserve l'intelligibilité géométrique de la scène et la localisation des contours, tandis que la **magnitude** détermine la distribution du contraste et les amplitudes globales.
- **Efficacité Algorithmique :** Le Théorème de Convolution permet le traitement de masques à grande échelle dans le domaine fréquentiel via la FFT, réduisant la complexité computationnelle asymptotique de $O(N^2K^2)$ dans l'espace à $O(N^2 \log N)$.
- **Phénomène de *Ringing* :** Les coupures abruptes dans le spectre (filtres idéaux) génèrent des oscillations spatiales indésirables (phénomène de Gibbs). L'atténuation douce par des filtres de **Butterworth** ou **gaussiens** élimine ces discontinuités.
- **Analyse Multirésolution via *Wavelets* :** Dépassant le caractère purement global de Fourier, la DWT capture simultanément la fréquence et la localisation spatiale, fondant le standard JPEG 2000 et appuyant des représentations hiérarchiques analogues aux extractions de caractéristiques dans les Réseaux de Neurones Convolutifs (CNN).
- **Compression Perceptuelle (DCT) :** Le *pipeline* JPEG exploite les limitations de contraste du système visuel humain aux hautes fréquences spatiales. La DCT isole l'énergie de blocs 8×8, permettant à la quantification d'éliminer les coefficients AC des détails fins sans préjudice perceptuel sévère.

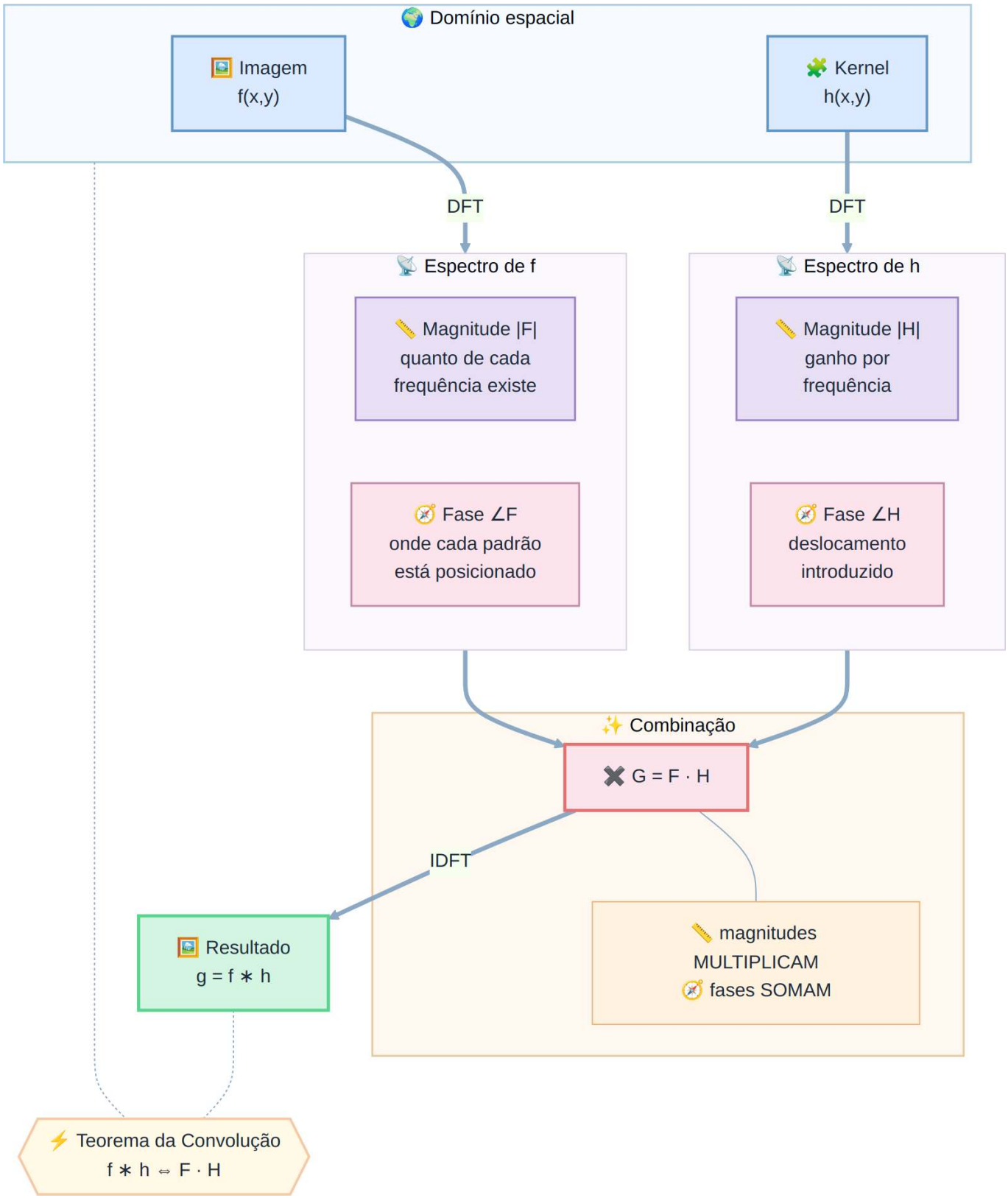


Figure 6.31: Carte conceptuelle des transformations et propriétés dans le domaine fréquentiel.

Prochaines Étapes : Le **Chapitre 6** inaugure la Partie II de l'ouvrage, appliquant les outils de traitement d'images à la résolution de problèmes réels d'inspection industrielle. Seront explorées des techniques de **segmentation et d'analyse de formes** pour la détection automatique de défauts sur les lignes de production — depuis l'identification de défauts superficiels sur les pièces jusqu'à la lecture *QRCode* sur les épreuves, consolidant le pont entre la théorie présentée dans la Partie I et les exigences pratiques de la vision par ordinateur.

6.11 🤖 Utilisation de Gemini Notebook comme Tuteur Complémentaire

Dans cette édition, l'utilisation de la plateforme **Gemini Notebook** est encouragée comme outil d'apprentissage complémentaire — **et non comme substitut** de la lecture attentive, de la résolution d'exercices ou de l'expérimentation pratique. Fondé sur des architectures d'intelligence artificielle, le système utilise exclusivement le matériel pédagogique et les documents fournis par l'auteur comme base de connaissances, garantissant que les réponses générées soient conceptuellement alignées sur le contenu programmatique et l'approche pédagogique adoptée tout au long de cet ouvrage.

Accès au Tuteur Intelligent

🚀 ACCÉDER À Gemini Notebook : CHAPITRE 05

🌐 **Langue et Langage de Programmation**

Le projet de ce chapitre dans Gemini Notebook a été construit uniquement avec le texte en **portugais** et les exemples de code en **Python**. Si vous étudiez à partir de l'édition en anglais ou en français, ou si vous suivez le parcours en C++, les réponses du tuteur peuvent ne pas correspondre exactement à la version que vous lisez.

Directives concernant le Contenu Généré par Intelligence Artificielle

Bien que les outils d'intelligence artificielle constituent des alliés efficaces dans le processus d'apprentissage et de révision, le contenu généré est sujet à des incohérences ou à des imprécisions techniques. Par conséquent, la consultation systématique de manuels, d'articles scientifiques et de sources académiques indexées est indispensable pour une validation rigoureuse des informations. Il est vivement recommandé d'exécuter et de modifier les exemples pratiques en Python fournis dans ce chapitre comme méthode principale de vérification expérimentale des résultats.

6.12 Liste d'exercices

1. **(10 %) Implémentation directe de la TFD 2D :** Implémentez analytiquement la Transformée de Fourier Discrète 2D (TFD) sans l'aide de fonctions natives de bibliothèques (comme `np.fft.fft2`), en utilisant strictement la formulation mathématique définie dans la Équation 6.1 pour une matrice de dimensions 16×16. Effectuez la validation numérique en comparant les coefficients générés avec les résultats de la fonction `np.fft.fft2`, en vous assurant que l'écart absolu maximal soit inférieur à 10^{-8}. Mesurez les temps d'exécution des deux méthodes et présentez une justification théorique pour la disparité observée en termes de complexité asymptotique.

2. **(15 %) Suppression du bruit périodique :** Ajoutez des interférences sinusoïdales avec des fréquences spatiales $(u_0, v_0) \in \{(5, 10), (20, 5), (30, 30)\}$ à l'image de test du *Cameraman*. Pour chaque scénario de dégradation, concevez un masque de filtrage *notch* spécifique dans le domaine fréquentiel afin d'isoler et d'atténuer les pics harmoniques indésirables. Évaluez quantitativement l'efficacité du processus de restauration par le calcul des métriques PSNR et SSIM. Discutez analytiquement du compromis entre l'atténuation du bruit sinusoïdal et l'atténuation indésirable des caractéristiques structurelles légitimes de l'image.

3. **(15 %) Analyse comparative des opérateurs passe-bas :** Réalisez une étude comparative entre les filtres passe-bas idéal, gaussien et de Butterworth (avec des ordres harmoniques $n = 1, 2, 4$), paramétrés avec des fréquences de coupure $D_0 = 20, 40, 60$ pixels. Pour chaque combinaison structurelle, calculez les indices PSNR et SSIM de l'image résultante par rapport au signal original de référence. Organisez les données quantitatives dans un tableau structuré et tracez les graphiques unidimensionnels des fonctions de transfert correspondantes le long du profil horizontal $H(u, 0)$.

4. **(15 %) Banque de filtres multirésolution de Haar :** Développez un script pour exécuter manuellement la décomposition *wavelet* discrète 2D de premier niveau en utilisant la famille de Haar. L'algorithme doit calculer les coefficients des filtres correspondants passe-bas (h) et passe-haut (g), en les appliquant de manière séparable sur les lignes et les colonnes de la matrice, suivis de l'opération de décimation (sous-échantillonnage spatial par un facteur de 2). Validez numériquement l'exactitude de votre implémentation en confrontant les sous-bandes obtenues avec la sortie de la fonction `pywt.dwt2(img, 'haar')`.

5. **(15 %) Compression éparse par seuillage wavelet :** Appliquez la technique de filtrage par seuillage abrupt (*hard thresholding*) sur les coefficients de détail de la décomposition *wavelet*, en adoptant les seuils numériques $T \in \{5, 10, 20, 40, 80\}$ pour les familles de Haar, Daubechies (`db4`) et Symlets (`sym4`). Après avoir réalisé le processus de synthèse au moyen de la transformée inverse (`pywt.waverec2`), calculez les valeurs de PSNR et SSIM de chaque image reconstruite. Identifiez et justifiez quelle combinaison de famille *wavelet* et de seuil T maximise la similarité structurelle.

6. **(15 %) Construction d'un encodeur JPEG simplifié :** Implémentez le pipeline complet de compression de données simulant la norme JPEG. Le flux doit englober : la conversion spatiale $RGB \to YC_bC_r$, le sous-échantillonnage chromatique dans la proportion 4:2:0, la segmentation de la luminance en blocs disjoints de 8×8 pixels, l'application de la DCT-II 2D orthogonale, et la quantification linéaire basée sur la matrice normalisée de luminance mise à l'échelle par des facteurs de qualité souhaités. Réalisez le décodage inverse et comparez quantitativement les reconstructions avec les fichiers générés par la fonction `cv2.imencode` pour les facteurs de qualité de 20, 50 et 80.

7. **(15 %) Analyse perceptuelle sur des contenus hétérogènes :** Développez une image synthétique composée de trois régions distinctes et aux caractéristiques spectrales contrastées : une texture photographique complexe (représentant de hautes fréquences stochastiques), une zone de texte vectorisé avec des bords nets (représentant des transitions en échelon pures) et un gradient linéaire continu (représentant de basses fréquences homogènes). Soumettez cette image mixte aux processus de compression sous les formats JPEG, PNG et WebP. Évaluez et interprétez les résultats en corrélant la taille finale du fichier sur disque avec les métriques PSNR et SSIM obtenues, en justifiant quel format présente les meilleures performances pour des signaux de nature hétérogène et pourquoi cet avantage survient en termes de compactage d'énergie et de préservation perceptuelle.

Références du chapitre

Le fondement théorique et le développement analytique des concepts abordés dans ce chapitre s'appuient sur les ouvrages de référence suivants :

- **Gonzalez (2018)** — Formulations classiques des Transformées de Fourier Discrètes 2D (TFD), conception de filtres analytiques dans le domaine fréquentiel, Transformée en Cosinus Discrète (TCD) et principes fondamentaux des systèmes de compression d'images.
- **Oppenheim (2010)** — Théorie formelle des signaux et des systèmes appliqués dans le domaine discret, couvrant les propriétés mathématiques de la TFD et la modélisation analytique du Théorème de Convolution.
- **Mallat (1999)** — Fondements mathématiques de la théorie des *ondelettes*, formalisation de l'analyse multirésolution (AMR) et architecture des bancs de filtres dyadiques.
- **Wallace (1991)** — Spécification originale et aspects techniques du standard de compression ISO/CEI JPEG, avec un accent sur les critères psychovisuels pour la conception des matrices de quantification TCD.

- **Szeliski (2022)** — Modélisation computationnelle et caractérisation des métriques modernes de fidélité et de qualité perceptuelle (PSNR et SSIM), ainsi que l'analyse comparative des formats d'images tramées à hautes performances.

Exécuter Colab Ouvrir GitHub

6.13 💻 Partie pratique avec exercices de programmation

La présente liste d'exercices de programmation (EP) consolide les formulations théoriques présentées tout au long du Chapitre 5 — Transformées et Compression — au moyen d'un parcours pratique appliqué. Les exercices sont structurés à partir de matrices de dimensions réduites, permettant la validation analytique et l'inspection manuelle de chaque coefficient, tout en maintenant la cohérence méthodologique adoptée dans les chapitres précédents.

L'enchaînement des exercices reproduit rigoureusement le flux conceptuel du chapitre : on commence par l'implémentation explicite de la Transformée de Fourier Discrète (TFD) à partir de sa définition mathématique fondamentale ; on progresse vers la conception de filtres passe-bas et de masques *notch* dans le domaine fréquentiel ; on applique la quantification des coefficients (noyau de la compression avec perte) ; et l'on conclut par l'intégration de ces étapes dans la construction d'un *pipeline* de compression JPEG simplifié et dans l'analyse perceptuelle des formats d'image.

! Directives pour la résolution des exercices de programmation

Dans tous les exercices de ce chapitre, les coordonnées du **centre du spectre** (origine des fréquences spatiales après application du décalage `fftshift`) doivent être déterminées par division entière. Pour une matrice de L lignes et C colonnes, la composante de fréquence nulle se situe à la position :

$$(c_y, c_x) = \left(\left\lfloor \frac{L}{2} \right\rfloor, \left\lfloor \frac{C}{2} \right\rfloor \right)$$

Cette convention est rigoureusement identique à celle adoptée par la fonction `np.fft.fftshift`. De plus, à toutes les étapes exigeant une discrétisation ou un arrondi numérique (que ce soit dans la quantification des coefficients AC ou dans la reconstruction finale des pixels), on doit employer l'arrondi standard au plus proche entier (*round half away from zero*), atténuant les ambiguïtés sur les valeurs dont la fraction est exactement égale à 0.5.

🎯 Objectif de ce cahier

Ce cahier permet de développer, valider, organiser et tester des solutions d'**Exercices de Programmation (EPs)** dans des environnements interactifs, comme Colab, avec les mêmes cas de test que Moodle, en les y copiant uniquement au moment d'enregistrer la note officielle.

Téléchargement

Téléchargez `morph.py` et `testsuite.py` en exécutant la cellule ci-dessous :

```
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
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0 | TestSuite : 1.1.2
```

Exécution des tests

Pour évaluer les tests, exécutez `TestSuite("EP05_01.extensão").run()` dans une nouvelle cellule, en remplaçant l'extension par celle du langage utilisé (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). Le système télécharge les cas de test depuis GitHub, exécute le programme et calcule automatiquement la note.

Pour tester directement du code Python, sans sauvegarder de fichier, utilisez `run_code(codigo)` en passant le code sous forme de *chaîne de caractères* dans une variable `codigo` :

```
codigo = """
from morph import mm
# ... votre code ici ...
"""
TestSuite("EP05_01").run_code(codigo)
```

6.13.1 EP05_01 🟢 Filtre Passe-Bas Idéal par Distance dans le Spectre

Dans un ***scanner* de documents anciens**, le capteur capte le papier froissé et la texture des fibres en même temps que le texte — un bruit haute fréquence qui « pollue » le spectre sur les bords. Le technicien de maintenance n'a pas accès à l'image originale, seulement au **spectre de magnitude déjà calculé** par le logiciel du *scanner*. Son travail est simple et chirurgical : ne conserver que le **cercle central** des basses fréquences (la structure globale du document) et effacer tout ce qui se trouve hors du rayon D_0, éliminant la texture fine sans même avoir à toucher à l'image spatiale.

C'est le **filtre passe-bas idéal (LPFI)** : l'opération spectrale la plus directe du chapitre, mais aussi celle qui révèle le mieux l'anatomie d'un spectre centré.

6.13.1.1 📋 Directives d'implémentation

1. **Dimensions :** Lire les entiers L (lignes) et C (colonnes) du spectre de magnitude — déjà fourni **centré** (équivalent à la sortie de `np.fft.fftshift`).
2. **Fréquence de coupure :** Lire l'entier D_0.
3. **Données :** Lire les valeurs entières de la matrice de magnitude, ligne par ligne.
4. **Centre du spectre :** Calculer $(c_y, c_x) = (L \,//\, 2,\ C \,//\, 2)$.
5. **Distance :** Pour chaque position (u, v), calculer
$$D(u,v) = \sqrt{(u - c_y)^2 + (v - c_x)^2}$$
6. **Masque idéal :** Appliquer
$$H(u,v) = \begin{cases} 1, & D(u,v) \le D_0 \\ 0, & D(u,v) > D_0 \end{cases}$$
7. **Filtrage :** La valeur de sortie est $\text{mag}'(u,v) = \text{mag}(u,v) \cdot H(u,v)$.
8. **Sortie :** Afficher la matrice filtrée avec les dimensions $L \times C$.

6.13.1.2 📌 Contraintes computationnelles

- **Comparaison non stricte :** le critère utilise $D(u,v) \le D_0$ (la frontière appartient au filtre, c'est-à-dire qu'elle est conservée).
- **Type :** toutes les valeurs d'entrée et de sortie sont des entiers ; la distance est calculée en virgule flottante uniquement en interne.
- **Pas d'arrondi de magnitude :** comme l'entrée est déjà entière et que le masque est binaire (0 ou 1), la sortie n'a jamais besoin d'arrondi.

6.13.1.3 🧠 Fondement théorique

Région	Distance au centre	Effet du filtre
Centre ($D \leq D_0$)	Basses fréquences	Préservées — structure globale conservée
Bords ($D > D_0$)	Hautes fréquences	Mises à zéro — texture et bruit supprimés
D_0 **petit**	—	L'image reconstruite serait très floue
D_0 **grand**	—	Peu de filtrage ; presque toute l'énergie est préservée

6.13.1.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Entier D_0.
- Lignes suivantes : Éléments entiers de la matrice de magnitude (centrée).

Sortie :

- Matrice filtrée en L lignes et C colonnes, séparés par des espaces.

6.13.1.5 📌 Exemples

Entrée	Sortie	Observation
3 3 1 10 20 30 40 50 60 70 80 90	0 20 0 40 50 60 0 80 0	Centre $(1, 1)$. Les coins ont $D = \sqrt{2} \approx 1.41 > 1$, donc ils sont mis à zéro ; les voisins orthogonaux ont $D = 1 \leq 1$ et sont conservés.
1 3 0 5 9 7	0 9 0	$L = 1, C = 3$: centre en $(0, 1)$. Seule la position centrale elle-même ($D = 0$) survit à $D_0 = 0$.

```
%%writefile EP05_01.py
# Code Python
```

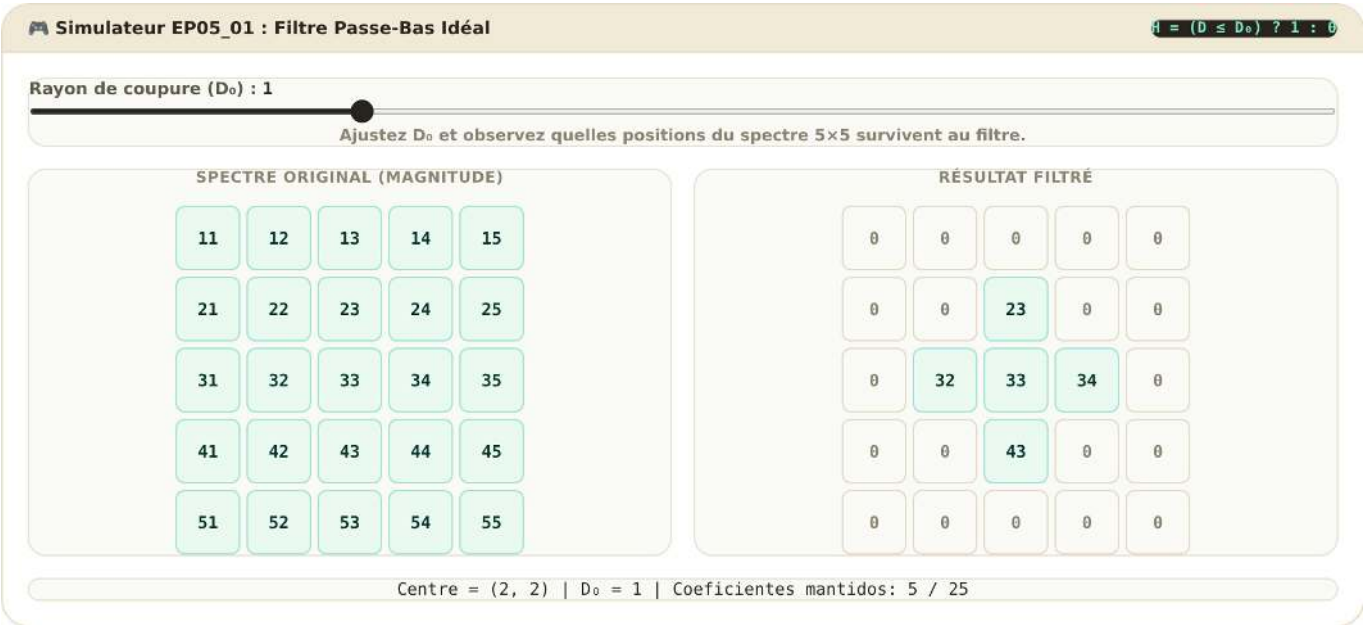
```
Overwriting EP05_01.py
```

```
TestSuite("EP05_01.py").run()
```

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

6.13.2 EP05_02 🟡 Filtre *Notch* : Suppression des pics périodiques

Une caméra d'**inspection industrielle** capture des images de circuits imprimés, mais l'alimentation électrique de la ligne de production introduit une **interférence électrique périodique** — un motif de stries quasi



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap05/sim-ep0501.html

Figure 6.32: Simulateur EP05_01 : Filtre passe-bas idéal dans le spectre

imperceptible à l'œil nu, mais qui apparaît dans le spectre de Fourier comme des **paires de pics brillants** symétriquement positionnés autour du centre. L'équipe de vision par ordinateur ne peut pas recapturer l'image : elle doit **localiser et effacer chirurgicalement** ces paires de pics dans le spectre, tout en préservant le reste de l'information utile de l'image.

C'est le rôle du **filtre coupe-bande *notch*** : contrairement au passe-bas (qui affecte une région continue), il cible **des points spécifiques et leurs symétriques**, laissant le reste du spectre intact.

6.13.2.1 📋 Directives d'implémentation

1. **Dimensions :** Lire les entiers L (lignes) et C (colonnes) du spectre de magnitude centré.
2. **Données :** Lire les valeurs entières de la matrice de magnitude, ligne par ligne.
3. **Pics :** Lire l'entier K (nombre de paires de pics à supprimer).
4. **Pour chacun des K pics :** lire trois entiers Δv, Δu, r — déplacement vertical, déplacement horizontal et rayon du *notch*.
5. **Centre du spectre :** $(c_y, c_x) = (L \,//\, 2,\ C \,//\, 2)$.
6. **Suppression symétrique :** pour chaque pic, mettre à zéro **toutes** les positions (u, v) telles que la distance au point $(c_y + \Delta v,\ c_x + \Delta u)$ soit $\leq r$, **et également** toutes les positions à distance $\leq r$ du point symétrique $(c_y - \Delta v,\ c_x - \Delta u)$.
7. **Sortie :** Afficher la matrice résultante avec les dimensions $L \times C$.

6.13.2.2 📌 Contraintes computationnelles

- **Symétrie obligatoire :** chaque pic fourni génère **deux** disques mis à zéro (le point et son symétrique par rapport au centre) — oublier le symétrique est l'erreur la plus courante.
- **Chevauchement :** si deux disques se chevauchent, la position reste à zéro (ni « addition » ni restauration).
- **Comparaison non stricte :** une position est mise à zéro si distance $\leq r$.
- **Ordre de lecture :** les K pics doivent être traités dans l'ordre où ils apparaissent en entrée, mais le résultat final est indépendant de l'ordre (les opérations de mise à zéro sont commutatives).

6.13.2.3 🧠 Fondement théorique

Concept	Rôle dans le filtre *notch*
Pic en $(\Delta v, \Delta u)$	Fréquence de l'interférence périodique détectée visuellement dans le spectre
Point symétrique $(-\Delta v, -\Delta u)$	Toute DFT d'un signal réel est hermitienne : les pics apparaissent toujours en paires symétriques par rapport au centre
Rayon r	Contrôle la « largeur » de la réjection — un r grand élimine davantage d'énergie autour du pic, mais aussi davantage d'information utile

6.13.2.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Lignes suivantes : Éléments entiers de la matrice de magnitude (centrée), L lignes.
- Ligne suivante : Entier K.
- K lignes suivantes : trois entiers Δv, Δu, r (séparés par des espaces).

Sortie :

- Matrice résultante sur L lignes et C colonnes, séparés par des espaces.

6.13.2.5 📌 Exemples

Entrée	Sortie	Observation
5 5 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 1 1 1 0	1 2 3 4 5 6 0 8 9 10 11 12 13 14 15 16 17 18 0 20 21 22 23 24 25	Centre $(c_y, c_x) = (2, 2)$. Pic fourni $(\Delta v, \Delta u) = (1, 1)$ génère le point $(3, 3)$ (valeur 19) et son symétrique $(1, 1)$ (valeur 7), tous deux mis à zéro avec $r = 0$ (seuls les points exacts).

```
%%writefile EP05_02.py
# Code Python
```

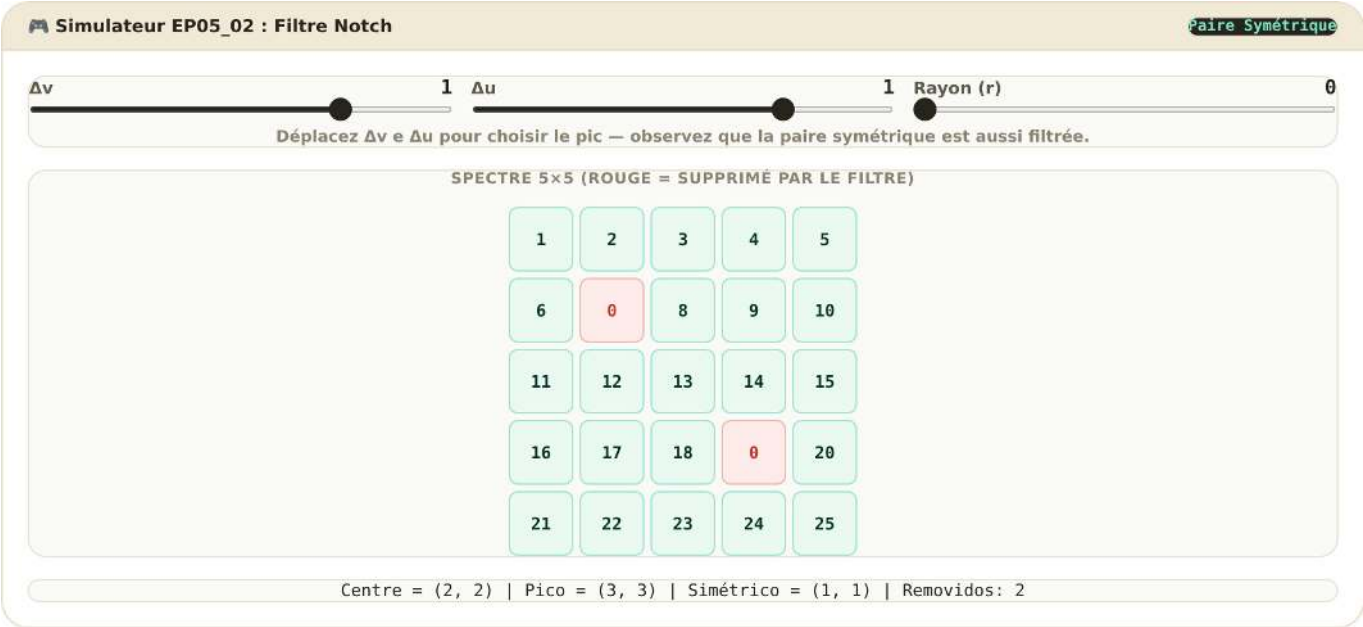
```
Overwriting EP05_02.py
```

```
TestSuite("EP05_02.py").run()
```

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

6.13.3 EP05_03 🟠 Quantification DCT : la véritable source de compression

Une application de **galerie de photos** doit réduire la taille de milliers d'images avant de les *téléverser* vers le cloud, sans tout recoder de zéro. L'ingénieur responsable dispose déjà des **coefficients DCT** de chaque bloc 4×4 calculés (l'étape coûteuse en calcul est déjà faite) — il ne reste plus qu'à appliquer la **table de quantification**, l'étape qui élimine réellement de l'information et génère la compression. Les coefficients



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap05/sim-ep0502.html

Figure 6.33: Simulateur EP05_02: Filtre Notch

haute fréquence, moins perceptibles à l'œil humain, reçoivent de grands diviseurs et tendent à devenir **zéro** ; les coefficients basse fréquence, plus perceptibles, reçoivent de petits diviseurs et survivent presque intacts.

Vous allez implémenter exactement cette étape : **quantifier et déquantifier** (diviser, arrondir, multiplier en retour) — le cœur de la compression *lossy* du JPEG.

6.13.3.1 📋 Directives d'implémentation

1. **Dimension du bloc :** Lire l'entier N (bloc $N \times N$).
2. **Coefficients :** Lire la matrice C des coefficients DCT, N lignes avec N entiers chacune (ils peuvent être négatifs).
3. **Table de quantification :** Lire la matrice Q, N lignes avec N entiers positifs chacune.
4. **Quantification :** Pour chaque position (u, v), calculer l'indice quantifié

$$\tilde{C}(u,v) = \text{round}\left(\frac{C(u,v)}{Q(u,v)}\right)$$

   en utilisant l'arrondi standard à l'entier le plus proche (les valeurs intermédiaires `.5` ne se produisent jamais dans les cas de test).
5. **Déquantification (reconstruction) :** Calculer

$$C'(u,v) = \tilde{C}(u,v) \times Q(u,v)$$

6. **Sortie :** Afficher la matrice reconstruite C', $N \times N$, entiers.

6.13.3.2 📌 Contraintes informatiques

- ***Aller-retour* complet :** la sortie est le coefficient **reconstruit** ($\tilde{C} \times Q$), pas l'indice quantifié isolé.
- **Division en virgule flottante :** la division $C(u,v)/Q(u,v)$ doit être effectuée en virgule flottante avant l'arrondi — une division entière tronquée produirait un résultat incorrect.
- **Signe préservé :** les coefficients négatifs conservent leur signe après quantification et reconstruction.
- $Q(u,v) > 0$ **toujours :** aucune gestion de division par zéro n'est nécessaire.

6.13.3.3 🧠 Fondements théoriques

Coefficient	Fréquence	Valeur typique de Q	Effet de la quantification
$C(0,0)$	DC (moyenne du bloc)	Petite	Survit presque toujours — domine l'énergie
$C(u,v)$ faible $u+v$	Basse fréquence	Petite/moyenne	Partiellement préservé
$C(u,v)$ élevé $u+v$	Haute fréquence	Grande	Devient souvent zéro — source de la compression

6.13.3.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

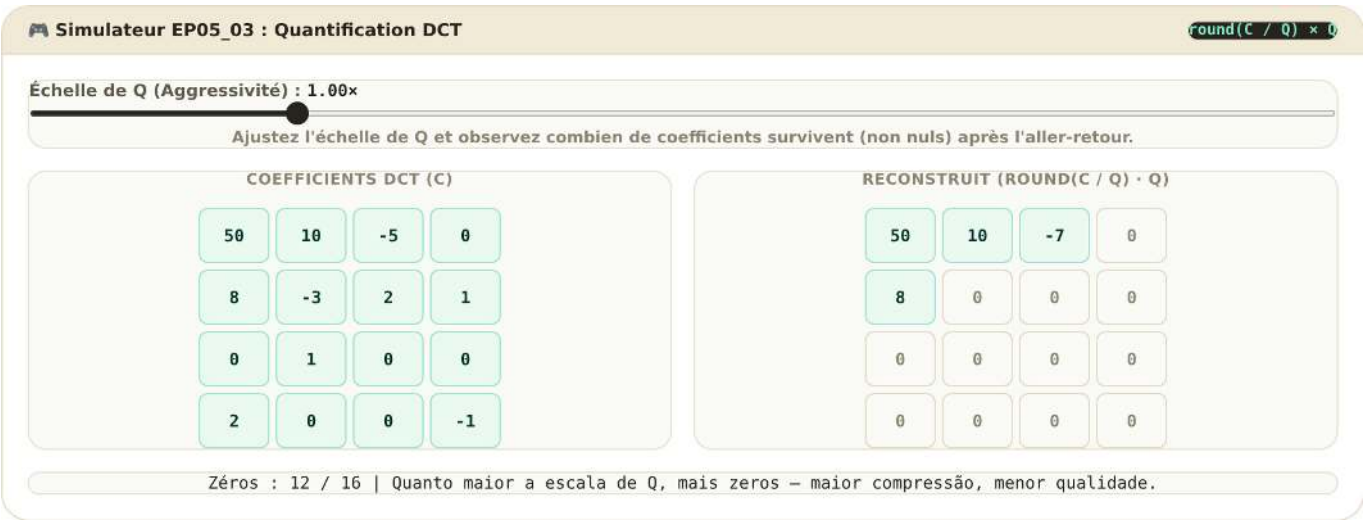
- Ligne 1 : Entier N.
- N lignes suivantes : matrice C (coefficients DCT, entiers, peuvent être négatifs).
- N lignes suivantes : matrice Q (table de quantification, entiers positifs).

Sortie :

- Matrice reconstruite C', $N \times N$, entiers séparés par des espaces.

6.13.3.5 📌 Exemples

Entrée	Sortie	Observation
4 50 10 -5 0 8 -3 2 1 0 1 0 0 2 0 0 -1 2 5 7 8 4 7 8 11 6 8 11 12 9 11 12 14	50 10 -7 0 8 0 0 0 0 0 0 0 0 0 0 0	$C(0,0) = 50/2 = 25 \to 25\times 2 = 50$ (préservé). $C(0,2) = -5/7 \approx -0.71 \to -1 \to -1 \times 7 = -7$. Quant à $C(1,1) = -3/7 \approx -0.43 \to 0$: mis à zéro par la quantification — la majeure partie du bloc devient zéro, illustrant la compaction de l'énergie dans le coin supérieur gauche.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap05/sim-ep0503.html

Figure 6.34: Simulateur EP05_03 : Quantification DCT (*round-trip*)

```
%%writefile EP05_03.py
# Code Python
```

```
Overwriting EP05_03.py
```

```
TestSuite("EP05_03.py").run()
```

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

6.13.4 EP05_04 🔴 Implémentation de la TFD 2D à partir de la définition

Un laboratoire de recherche en **astronomie computationnelle** a reçu, d'une mission ancienne, un petit capteur expérimental dont les données brutes ne peuvent pas être traitées par les bibliothèques modernes de FFT — l'environnement de validation est isolé et ne permet que des opérations arithmétiques de base. L'équipe doit **réimplémenter la Transformée de Fourier Discrète 2D à partir de la définition mathématique elle-même**, cellule par cellule, pour ensuite comparer bit à bit avec `np.fft.fft2` dans un autre environnement.

C'est l'exercice le plus conceptuel de la liste : il n'y a pas de raccourcis. Vous allez implémenter la double sommation de la Équation 6.1 directement, en mettant en évidence *pourquoi* la FFT existe — et le coût computationnel qu'elle évite.

6.13.4.1 📋 Directives d'implémentation

1. **Dimensions :** Lire les entiers M (lignes) et N (colonnes) de l'image $f(x, y)$.
2. **Données :** Lire les valeurs entières de $f(x, y)$, ligne par ligne.
3. **TFD 2D :** Pour chaque paire de fréquences (u, v) avec $u = 0, \dots, M-1$ et $v = 0, \dots, N-1$, calculer

$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y)\, e^{-j2\pi\left(\frac{ux}{M} + \frac{vy}{N}\right)}$$

   en utilisant l'identité d'Euler $e^{-j\theta} = \cos(\theta) - j\sin(\theta)$ pour séparer les parties réelle et imaginaire — **n'utilisez aucune fonction de FFT prête à l'emploi**.
4. **Magnitude :** Calculer $|F(u, v)| = \sqrt{\mathrm{Re}(F)^2 + \mathrm{Im}(F)^2}$ et arrondir à l'entier le plus proche.
5. **Sortie :** Afficher la matrice des magnitudes arrondies, $M \times N$, dans le même ordre (sans `fftshift` — la composante DC reste en $(0, 0)$).

6.13.4.2 📌 Contraintes computationnelles

- **Interdiction d'utiliser des bibliothèques de FFT :** l'implémentation doit calculer les double sommations explicitement (boucles imbriquées), même si c'est plus lent.
- **Sans `fftshift` :** la sortie conserve la convention brute de la TFD, avec la composante DC en $F(0, 0)$ (coin supérieur gauche).
- **Arrondi :** la magnitude finale doit être arrondie à l'entier le plus proche ; dans les cas de test, il n'y a pas d'ambiguïté `.5`.
- **Précision :** de petites erreurs de virgule flottante (de l'ordre de 10^{-6}) avant l'arrondi sont attendues et n'affectent pas le résultat entier final.

6.13.4.3 🧠 Fondement théorique

Élément	Signification
$F(0, 0)$	Composante DC — somme de tous les pixels, $F(0, 0) = \sum f(x, y)$
Partie réelle $\mathrm{Re}(F)$	Projection du signal sur les cosinus

Élément	Signification
Partie imaginaire $\mathrm{Im}(F)$	Projection du signal sur les sinus
Complexité de cette implémentation	$\mathcal{O}((MN)^2)$ — c'est pourquoi la FFT, avec $\mathcal{O}(MN\log(MN))$, est indispensable pour les images réelles

6.13.4.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

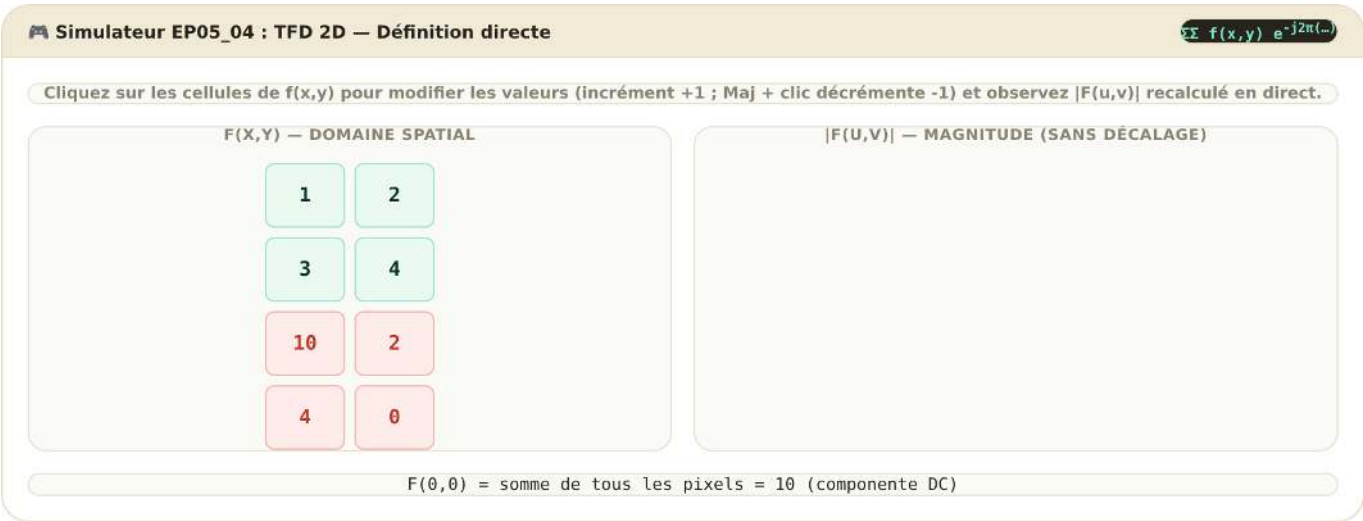
- Ligne 1 : Entier M.
- Ligne 2 : Entier N.
- Lignes suivantes : Éléments entiers de $f(x, y)$, M lignes.

Sortie :

- Matrice des magnitudes $|F(u, v)|$ arrondies, $M \times N$, séparées par des espaces.

6.13.4.5 📌 Exemples

Entrée	Sortie	Remarque
2 2 1 2 3 4	10 2 4 0	$F(0,0) = 1+2+3+4 = 10$ (DC = somme totale). $F(0,1) = (1-2)+(3-4) = -2 \to \lvert F\rvert = 2$. $F(1,0) = (1+2)-(3+4) = -4 \to \lvert F\rvert = 4$. $F(1,1) = (1-2)-(3-4) = 0$.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap05/sim-ep0504.html

Figure 6.35: Simulateur EP05_04: DFT 2D manuel

```
%%writefile EP05_04.py
# Code Python
```

```
Overwriting EP05_04.py
```

```
TestSuite("EP05_04.py").run()
```

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

6.13.5 EP05_05 🏆 *Pipeline* JPEG complet : DCT, quantification et reconstruction

Vous avez été chargé de créer, à partir de zéro, un **codec JPEG didactique** dans un environnement embarqué, sans aucune bibliothèque d'image disponible — uniquement des opérations mathématiques de base. Le client veut comprendre exactement où la qualité est perdue et où elle est récupérée, bloc par bloc. C'est le défi final du chapitre : intégrer **tout** ce qui a été étudié — la DCT-II orthonormale, la quantification perceptuelle et la reconstruction via IDCT — dans un seul *pipeline* de bout en bout, traitant un bloc $N \times N$ du début à la fin, exactement comme le fait le standard JPEG en interne, 8×8 pixels à la fois.

6.13.5.1 📋 Directives d'implémentation

1. **Dimension du bloc :** Lire l'entier N.
2. **Bloc original :** Lire la matrice de pixels $f(x, y)$, N lignes avec N entiers dans $[0, 255]$.
3. **Table de quantification :** Lire la matrice Q, $N \times N$ entiers positifs.
4. **Centrage :** Soustraire 128 de chaque pixel : $g(x, y) = f(x, y) - 128$.
5. **DCT-II 2D orthonormale :** Calculer

$$C(u, v) = \alpha(u)\,\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} g(x, y)\, \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right]$$

   avec $\alpha(0) = \sqrt{1/N}$ et $\alpha(k) = \sqrt{2/N}$ pour $k > 0$.
6. **Quantification :** $\tilde{C}(u, v) = \text{round}(C(u, v)/Q(u, v))$.
7. **Déquantification :** $C'(u, v) = \tilde{C}(u, v) \times Q(u, v)$.
8. **IDCT-II 2D (inverse orthonormale) :** Calculer $g'(x, y)$ à partir de $C'(u, v)$ en utilisant la transformée inverse correspondante (même base, somme sur u, v).
9. **Inversion du centrage et arrondi :** $f'(x, y) = \text{round}(g'(x, y) + 128)$, restreint à l'intervalle $[0, 255]$ (*clipping*).
10. **Sortie :** Afficher le bloc reconstruit f', $N \times N$, entiers.

6.13.5.2 📌 Contraintes computationnelles

- ***Pipeline* complet obligatoire :** toutes les six étapes (centrer, DCT, quantifier, déquantifier, IDCT, inverser) doivent être implémentées — sauter la quantification ne réussit pas les tests, car le résultat serait identique à l'original.
- ***Clipping* :** les valeurs reconstruites hors de $[0, 255]$ doivent être tronquées (0 si négatif, 255 si supérieur à 255).
- **Arrondi :** à la fois dans la quantification et dans la reconstruction finale des pixels, utilisez un arrondi standard ; les cas de test évitent toute ambiguïté `.5`.
- **Base orthonormale :** la normalisation $\alpha(u)$ et $\alpha(v)$ doit être appliquée exactement comme spécifié — sans elle, l'IDCT ne reconstruit pas correctement.

6.13.5.3 🧠 Fondement théorique

Étape	Analogue dans le standard JPEG réel	Où la qualité est perdue
Centrage	Identique — la DCT suppose un signal centré sur zéro	Aucune perte
DCT-II	Étapes 3–4 du *pipeline* (Table 6.7)	Aucune perte (transformation exacte et réversible)

Étape	Analogue dans le standard JPEG réel	Où la qualité est perdue
Quantification	Étape 5 — division par $Q(u, v)$	**Principale source de perte** — les coefficients de haute fréquence deviennent zéro
IDCT	Reconstruction finale	Reconstruit exactement les coefficients *quantifiés*, pas les originaux

6.13.5.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entier N.
- N lignes suivantes : bloc original $f(x, y)$, entiers dans $[0, 255]$.
- N lignes suivantes : table de quantification Q, entiers positifs.

Sortie :

- Bloc reconstruit $f'(x, y)$, $N \times N$, entiers dans $[0, 255]$, séparés par des espaces.

6.13.5.5 📌 Exemples

Entrée	Sortie	Observation
4 120 130 125 128 115 140 135 122 118 150 160 130 110 120 145 138 4 6 8 10 6 8 10 12 8 10 12 16 10 12 16 20	118 126 119 131 114 143 140 119 117 149 159 130 107 121 146 139	Après DCT, quantification agressive des hautes fréquences (grandes valeurs de Q en bas à droite) et reconstruction via IDCT, le bloc reste **proche** de l'original, mais pas identique — la différence est le coût de la compression *lossy*.

6.13.5.6 💡 Conseil de débogage

Si le résultat ne correspond pas, vérifiez dans cet ordre : (1) les coefficients DCT bruts (avant quantification) — ils doivent reconstruire l'original **exactement** via IDCT si vous sautez les étapes 6–7 ; (2) la table $\alpha(u)$ — erreur courante : appliquer $\sqrt{2/N}$ aussi pour $u = 0$; (3) l'arrondi de la quantification, qui doit se produire **avant** de multiplier à nouveau par Q.

```
%%writefile EP05_05.py
# Code Python
```

```
Overwriting EP05_05.py
```

```
TestSuite("EP05_05.py").run()
```

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

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap05/sim-ep0505.html

Figure 6.36: Simulateur EP05_05 : *Pipeline* JPEG complet par bloc

Partie II

Partie II — Vision par Ordinateur

Chapitre 7

Analyse de documents et inspection industrielle

Exécuter Colab Ouvrir GitHub

Dans la **Partie I — Traitement numérique des images (TNI)**, ont été étudiées des techniques de transformation et d'amélioration des images, telles que les opérations morphologiques, le filtrage spatial, les convolutions, le seuillage, la segmentation et le traitement dans le domaine fréquentiel.

La **Partie II — Vision par ordinateur (VO)** élargit ce champ en abordant l'interprétation automatique du contenu visuel, impliquant l'extraction d'informations, la reconnaissance de formes et la prise de décisions à partir d'images.

Ce chapitre présente cette transition à travers deux applications représentatives :

1. **Analyse automatisée de documents**, appliquée au traitement de formulaires, d'évaluations et d'autres documents structurés au moyen de systèmes de reconnaissance optique de marques (*Optical Mark Recognition* – OMR) ;
2. **Inspection industrielle automatisée**, axée sur le contrôle qualité et la détection de défauts sur les chaînes de production.

Ces applications intègrent des techniques de détection de structures géométriques, d'extraction de descripteurs invariants, de reconnaissance de formes et de classification d'objets, constituant la base de nombreux systèmes modernes d'inspection visuelle et d'automatisation.

7.1 Objectifs du chapitre

À la fin de ce chapitre, l'étudiant devrait être capable de :

- **Évaluer l'influence du prétraitement** sur la qualité de la reconnaissance automatique des informations dans les documents ;
- **Effectuer la reconnaissance optique de caractères** (ROC) pour convertir des documents numérisés en texte codé ;
- **Appliquer des techniques de traitement du langage naturel**, y compris la traduction automatique, au texte obtenu par ROC ;
- Appliquer des techniques d'**alignement et de rectification géométrique de documents** utilisant la transformée de Hough et les transformations projectives ;
- **Implémenter des systèmes de reconnaissance optique de marques** (ROM) pour la lecture automatisée d'évaluations et de formulaires ;
- **Détecter et segmenter des régions d'intérêt** sur la base d'opérations morphologiques, de contours et de propriétés géométriques ;
- **Décoder des marqueurs bidimensionnels et des codes-barres**, en intégrant des bibliothèques de vision par ordinateur aux *pipelines* de traitement documentaire ;
- **Développer des *pipelines* de vision par ordinateur** pour l'analyse automatisée de documents.

Ce chapitre marque la transition du **traitement numérique des images**, axé sur la transformation des images, vers la **vision par ordinateur**, dont l'objectif est d'interpréter le contenu visuel, d'extraire des informations et de soutenir des processus automatisés d'analyse et de prise de décision.

7.2 Configuration de l'environnement

Les exemples de ce chapitre utilisent des bibliothèques largement employées en PDI-VC. Le bloc ci-dessous installe les paquets nécessaires ; dans les environnements qui les possèdent déjà, l'exécution peut être ignorée.

```
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

# installer plus de dépendances en plus de morph.py pour ce chapitre
import sys, subprocess, importlib, shutil

def setup_cap06():
    """Installe les dépendances système et Python spécifiques au chapitre 6
    (OCR, lecture de PDF, code-barres)."""

    # 1. Dépendances système
    if 'google.colab' in sys.modules:
        print("[ENVIRONNEMENT] Google Colab. Configuration des dépendances système...")
        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:
        # Environnement local sans tesseract : tentative d'installation via apt-get (nécessite
        sudo/root)
        if shutil.which("apt-get"):
            print("[ENVIRONNEMENT] Local. Installation de tesseract-ocr via apt-get (peut
            demander un mot de passe)...")
            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(
                    "[AVERTISSEMENT] Impossible d'installer automatiquement. "
                    "Installez manuellement : sudo apt install tesseract-ocr
                    tesseract-ocr-por"
                )
        else:
            print(
                "[AVERTISSEMENT] tesseract introuvable et apt-get indisponible. "
                "Installez manuellement avant d'exécuter les cellules OCR."
            )

    # 2. Dépendances Python (installe uniquement celles manquantes)
    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"[AVERTISSEMENT] Échec de l'installation de {pkg} (nécessaire pour le
                module {mod}).")


setup_cap06()

# 3. Imports globaux du pipeline
import cv2, numpy as np, matplotlib.pyplot as plt
from skimage import io, data, color
```

```
 Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0
[ENVIRONNEMENT] Local. Installation de tesseract-ocr via apt-get (peut demander un mot de
passe)...
[AVERTISSEMENT] Impossible d'installer automatiquement. Installez manuellement : sudo apt
install tesseract-ocr tesseract-ocr-por
```

Outre ces bibliothèques, on utilisera le module didactique `morph.py`, développé pour simplifier les opérations de lecture, de visualisation et de traitement d'images tout au long de cet ouvrage. Le code suivant vérifie sa disponibilité, effectue le *téléchargement* si nécessaire et confirme la version chargée.

```
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" Environnement prêt. morph {getattr(morph, '__version__', 'local_file')}")
```

```
 Environnement prêt. morph 1.1.9
```

7.3 Bases d'images pour l'expérimentation

Les exemples présentés dans cette partie de l'ouvrage utilisent, chaque fois que possible, des documents numérisés, des feuilles de réponses, des codes-barres, des *QRCodes* et d'autres images provenant d'applications réelles. Afin de rendre les expériences entièrement reproductibles — y compris dans des environnements sans accès à Internet ou aux fichiers originaux du MCTest —, on emploie également des images publiques largement utilisées dans l'enseignement et la recherche en Traitement Numérique des Images et Vision par Ordinateur (TNI-VO).

Pourquoi utiliser une base d'images de *benchmark* ?

Des images comme `camera()` et `coins()` sont employées depuis des décennies dans les manuels, les articles scientifiques et les supports pédagogiques de TNI-VO. Leur utilisation offre d'importants avantages :

- **Reproductibilité :** tout lecteur obtient exactement les mêmes images, quel que soit l'ordinateur ou le système d'exploitation utilisé, sans avoir besoin de *téléchargements* externes ni de fichiers spécifiques à cet ouvrage ;
- **Comparabilité :** les résultats produits peuvent être directement comparés à ceux rapportés dans la littérature, puisque les mêmes ensembles d'images sont largement adoptés comme référence ;
- **Concentration sur les algorithmes :** étant compactes, bien documentées et distribuées sans restrictions d'usage à des fins éducatives et scientifiques, ces images permettent de concentrer l'attention sur les techniques de traitement, en réduisant les interférences liées à l'acquisition ou à la gestion des données.

7.3.1 Images publiques avec `skimage.data`

Le module `skimage.data` met à disposition une collection d'images de référence largement utilisée dans les activités d'enseignement, de recherche et de validation d'algorithmes en PDI-VC. L'inspection de l'attribut `data.__all__` montre que la version actuelle rassemble **42 éléments**, incluant des photographies naturelles, des documents numérisés, des images médicales, de la microscopie, des textures, des motifs synthétiques, des modèles tridimensionnels et des séquences temporelles. Il convient de noter que certains de ces éléments correspondent à des fonctions utilitaires, telles que `data_dir()` et `download_all()`, et non à des images proprement dites.

Comme l'objectif de ce chapitre est de présenter des applications d'analyse documentaire et d'inspection visuelle, la Table 7.2 rassemble une **sélection représentative** des images les plus pertinentes, organisée selon leurs principales applications en Vision par Ordinateur. La Figure 7.1 présente un échantillon de ces images, regroupées selon la même classification adoptée dans le tableau.

Images utilisées dans ce chapitre

Bien que `skimage.data` mette à disposition des dizaines d'images de référence, seules quatre sont employées directement dans les expériences de ce chapitre. Elles ont été sélectionnées parce qu'elles reproduisent, de manière contrôlée, des caractéristiques fréquemment rencontrées dans les documents numérisés et dans les systèmes d'inspection visuelle industrielle. La Table 7.1 résume le rôle de chacune d'elles tout au long de ce chapitre.

Tableau 7.1: Images de skimage.data utilisées dans les expériences de ce chapitre.

Image	Application dans le chapitre
`data.page()`	Page numérisée utilisée dans les expériences de correction d'éclairage, de rehaussement local (CLAHE) et de seuillage automatique par Otsu.
`data.text()`	Document contenant du texte imprimé, employé pour illustrer la segmentation, l'extraction de contours et les étapes typiques d'OCR et d'OMR.
`data.coffee()`	Photographie en couleur présentant des variations naturelles d'éclairage, utilisée pour illustrer des techniques applicables à des scènes réelles non documentaires.

`data.brick()`	Texture de référence employée dans des exemples d'inspection de surface et de détection de défauts par analyse de variance locale.

Tableau 7.2: Sélection d'images publiques représentatives disponibles dans le module *skimage.data*, organisées par domaine d'application.

Categoria	Função	Descrição
📄 Documentos & OCR/OMR	`data.page()`	Página digitalizada de documento — normalização de fundo, CLAHE e Otsu.
📄 Documentos & OCR/OMR	`data.text()`	Texto impresso — segmentação e extração de contornos em cenários de OCR/OMR.
🔵 Segmentação, Morfologia & Contornos	`data.coins()`	Conjunto de moedas — referência clássica para segmentação e *watershed*.
🔵 Segmentação, Morfologia & Contornos	`data.clock()`	Relógio analógico — detecção de formas e contornos.
🔵 Segmentação, Morfologia & Contornos	`data.binary_blobs()`	Blobs binários sintéticos — conectividade e morfologia matemática.
🔵 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.

```
import matplotlib.pyplot as plt
from skimage import data

# Images triées par catégorie ; la couleur du titre reproduit la couleur de la catégorie dans
le tableau précédent
imgs = [
    ("page",         data.page(),         "#2563eb"),   # Documents & OCR/OMR
    ("text",         data.text(),         "#2563eb"),
    ("coins",        data.coins(),        "#16a34a"),   # Segmentation & morphologie
    ("binary_blobs", data.binary_blobs(), "#16a34a"),
    ("brick",        data.brick(),        "#ea580c"),   # Texture & inspection industrielle
    ("checkerboard", data.checkerboard(),"#ea580c"),
    ("camera",       data.camera(),       "#7c3aed"),   # Photographies classiques de
TNI/Vision par ordinateur
    ("coffee",       data.coffee(),       "#7c3aed"),
]

fig, ax = plt.subplots(2, 4, figsize=(11, 5.5))
for a, (nome, img, cor) in zip(ax.ravel(), imgs):
```

```
    a.imshow(img, cmap="gray")
    a.set_title(nome, color=cor, fontweight="bold")
    a.axis("off")
plt.tight_layout()
```

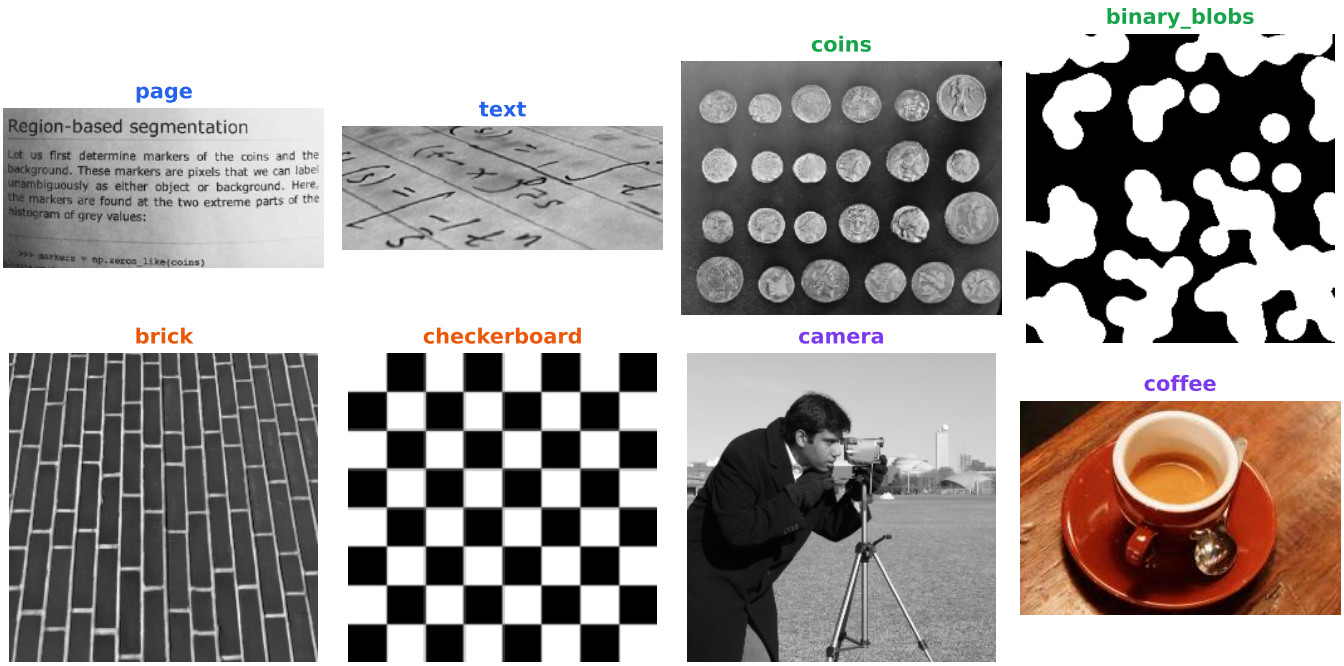


Figure 7.1: Échantillon d'images publiques de *skimage.data*, groupées par domaine d'application.

7.4 Normalisation du fond et égalisation locale de contraste

La qualité de la segmentation dépend directement des caractéristiques de l'image d'entrée. Dans les documents numérisés, les variations d'éclairage, les ombres, les régions surexposées et les différences de tonalité du papier réduisent le contraste entre le premier plan et le fond, rendant difficile l'application de méthodes de seuillage global, comme l'algorithme d'Otsu.

Pour minimiser ces effets, deux techniques complémentaires de prétraitement sont employées :

- **Normalisation du fond :** elle estime la composante basse fréquence de l'image au moyen d'un fort lissage, puis normalise l'image originale par rapport à ce fond estimé. Cette procédure réduit les gradients d'éclairage et compense les variations lentes d'intensité, tout en préservant les structures d'intérêt.
- **CLAHE** (*Contrast Limited Adaptive Histogram Equalization*), présenté au **Chapitre 4 :** il divise l'image en petites régions (*tiles*) et procède à l'égalisation de l'histogramme de chaque région de manière indépendante. Le contraste est limité afin d'éviter l'amplification excessive du bruit, ce qui rend la technique particulièrement adaptée aux images présentant des variations locales d'éclairage.

La Figure 7.2 compare ces stratégies en utilisant l'image `page()` de la bibliothèque `skimage.data`. Six résultats sont présentés : (a) l'image originale ; (b) la binarisation directe par la méthode d'Otsu, utilisée comme référence ; (c) le fond estimé par filtrage gaussien ; (d) l'image après la normalisation du fond ; (e) la binarisation obtenue après application du CLAHE suivie de la méthode d'Otsu ; et (f) la binarisation obtenue après normalisation du fond suivie de l'application de la méthode d'Otsu.

La comparaison permet d'observer l'effet produit par chaque étape du prétraitement et son influence sur la qualité de la segmentation. En particulier, la normalisation du fond réduit les variations globales d'éclairage, tandis que le CLAHE augmente le contraste local entre les caractères et le fond. Selon les caractéristiques de l'image, l'une ou l'autre stratégie peut produire des résultats supérieurs, aucune technique n'étant

universellement plus adaptée.

```
import cv2
from skimage import data
from morph import mm

img = data.page()  # ou : img = mm.gray(img_final) - avec l'image de la feuille d'examen

#   Méthode 1 : Normalisation du fond + Otsu
# Estime le fond avec un filtre gaussien à grand sigma (variations lentes de lumière)
# et divise pixel par pixel pour annuler le gradient d'éclairage
bg = cv2.GaussianBlur(img, (0, 0), sigmaX=25)
img_norm = cv2.divide(img, bg, scale=255)
img_norm_otsu = mm.threshold(img_norm)

#   Méthode 2 : CLAHE + Otsu
# tileGridSize définit la taille de chaque région locale (tuile)
# clipLimit contrôle le plafond d'amplification - des valeurs élevées augmentent le contraste
# mais amplifient aussi le bruit
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
img_clahe = clahe.apply(img)
img_clahe_otsu = mm.threshold(img_clahe)

#   Méthode 0 : Otsu direct (sans prétraitement) - référence
img_otsu = mm.threshold(img)

mm.show(
    [img,  img_otsu, bg, img_norm, img_clahe_otsu, img_norm_otsu],
    titles=["(a) Originale", "(b) Otsu direct", "(c) Gaussien",  \
            "(d) Fond normalisé (img/bg)", "(e) CLAHE + Otsu", \
            "(f) Normalisation du fond + Otsu"],
    cols=3,
    figsize=(14, 8)
)
```

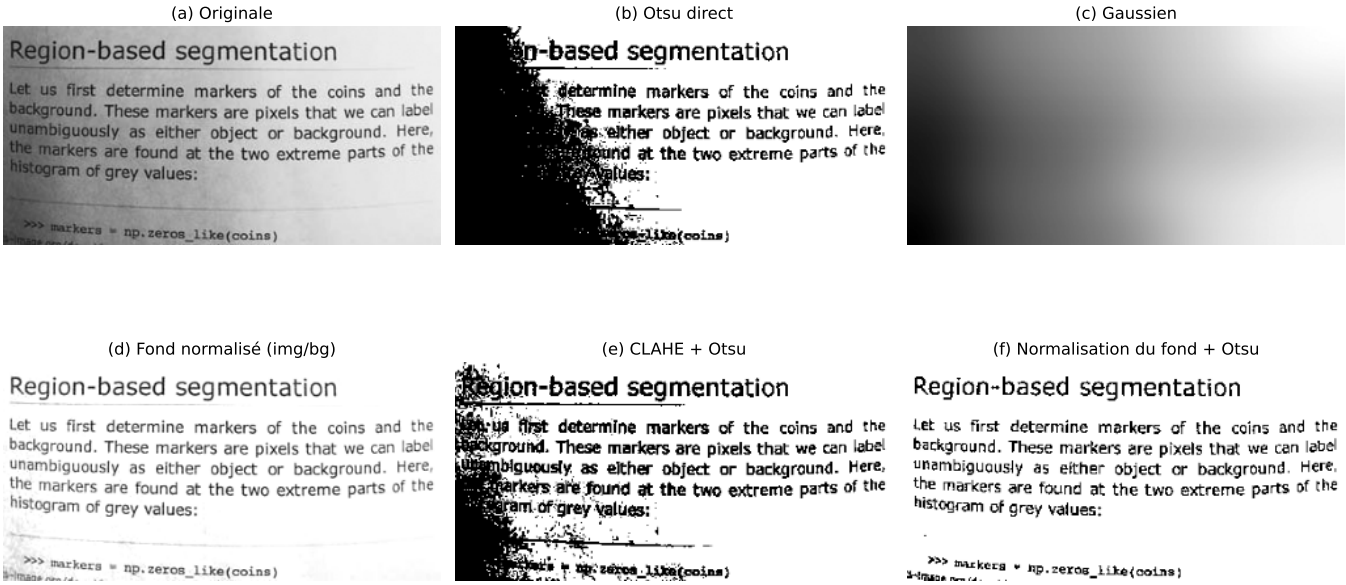


Figure 7.2: Comparaison entre stratégies de prétraitement pour la binarisation de l'image de texte : (a) image originale ; (b) seuillage direct par la méthode d'Otsu ; (c) fond estimé par filtrage gaussien ; (d) image après normalisation du fond (division par l'image lissée) ; (e) CLAHE suivi d'un seuillage par Otsu ; et (f) normalisation du fond suivie d'un seuillage par Otsu. Les images illustrent l'effet de chaque technique sur la compensation des variations d'éclairage et sur la qualité de la segmentation.

🧠 Pourquoi cela fonctionne-t-il ? — Normalisation du fond vs CLAHE

Normalisation du fond : en divisant l'image par une version fortement lissée d'elle-même, on élimine les variations lentes de luminosité (gradient de lumière, ombre de bord) sans affecter les détails fins — texte, lignes, bulles. Le résultat est une image à l'éclairage approximativement uniforme, où le seuil global d'Otsu fonctionne désormais correctement sur toute la page.
CLAHE : un histogramme global égalisé « étire » les tons de toute l'image d'un coup — utile lorsque l'éclairage est uniforme, mais problématique lorsqu'il ne l'est pas. Le CLAHE divise l'image en petits blocs (*tiles*) et égalise chacun séparément, avec une limite maximale d'amplification (`clipLimit`) pour ne pas exploser le bruit. Il est particulièrement efficace pour rehausser les régions sous-exposées localement, mais il n'élimine pas les gradients globaux — c'est pourquoi l'appliquer après la normalisation du fond tend à produire des résultats plus cohérents.

7.5 Reconnaissance Optique de Caractères (OCR)

Après la binarisation, l'étape suivante du traitement documentaire consiste à convertir la représentation visuelle des caractères en texte codé numériquement, processus appelé **Reconnaissance Optique de Caractères** (*Optical Character Recognition* — **OCR**).

De manière générale, un système d'OCR comprend trois étapes :

1. **Segmentation :** identifie les lignes, les mots et les caractères dans l'image, en utilisant des projections horizontales et verticales ou la détection de composantes connexes.
2. **Extraction de caractéristiques :** représente chaque caractère par des attributs visuels, tels que les bords, les courbures et les motifs de tracé.
3. **Reconnaissance :** associe les attributs extraits au caractère le plus probable. Les systèmes actuels utilisent principalement des réseaux neuronaux récurrents (LSTM) ou des architectures basées sur les *transformers*.

Dans cet ouvrage, on utilise le **Tesseract OCR**, accessible via la bibliothèque `pytesseract` (installation : `pip install pytesseract`). Développé à l'origine par Hewlett-Packard entre 1985 et 1995 et actuellement maintenu par Google, Tesseract est décrit dans Smith (2007) et Smith (2013). Dans les versions récentes, la reconnaissance textuelle est réalisée par des réseaux neuronaux LSTM.

Les performances de l'OCR dépendent de la qualité de l'image d'entrée. Le bruit, le faible contraste, les distorsions géométriques et l'éclairage irrégulier réduisent le taux de reconnaissance. Pour cette raison, des étapes telles que la rectification, la normalisation du fond et l'égalisation adaptative (CLAHE) intègrent le prétraitement de l'image.

🧠 Le Tesseract a-t-il besoin d'une image binarisée ?

Le Tesseract intègre en interne une étape de binarisation adaptative avant la reconnaissance des caractères. Pour cette raison, fournir à l'OCR une image préalablement binarisée ne produit pas toujours les meilleurs résultats.
Comme le seuillage est une opération irréversible, il peut éliminer les variations subtiles d'intensité sur les bords des caractères, comme l'*anticrénelage*, qui peuvent aider le mécanisme de reconnaissance. Dans de nombreux cas, une image en niveaux de gris, avec un bon éclairage et un bon contraste, produit une transcription plus fidèle que sa version binarisée.

Pour étudier cet effet, on compare le texte extrait par Tesseract à partir de quatre versions de la même image, présentées dans la Figure 7.3: (a) image originale ; (b) image soumise à l'égalisation adaptative (CLAHE) suivie d'un seuillage par la méthode d'Otsu ; (c) image soumise à la normalisation du fond suivie d'un seuillage par la méthode d'Otsu ; et (d) image soumise uniquement à la normalisation du fond, en préservant les niveaux de gris.

La comparaison entre les versions (c) et (d) montre que, dans les passages contenant des caractères visuellement similaires, la version en niveaux de gris (d) a produit une transcription plus fidèle au texte original que la version binarisée (c). Ce résultat indique que le seuillage appliqué lors du prétraitement peut éliminer des informations utiles à la reconnaissance. Ainsi, bien que la binarisation soit essentielle pour diverses opérations de traitement d'images, elle ne constitue pas nécessairement la meilleure entrée pour l'OCR. Le choix de la technique de prétraitement doit tenir compte de l'étape suivante du *pipeline* documentaire.

```
import pytesseract
import shutil as _sh
if _sh.which("tesseract") is None:
    # Environnement de build sans Tesseract installé (ex. : sans apt/sudo) :
    # dégrade au lieu de casser le rendu. Sur Colab/local avec Tesseract,
    # rien ne change.
    _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()

#   Réutilisation des résultats de la section précédente
img_otsu = mm.threshold(img)

bg = cv2.GaussianBlur(img, (0, 0), sigmaX=25)
img_norm = cv2.divide(img, bg, scale=255)       # niveaux de gris, sans Otsu
img_norm_otsu = mm.threshold(img_norm)          # binarisée

clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
img_clahe = clahe.apply(img)
img_clahe_otsu = mm.threshold(img_clahe)

#   Configuration de Tesseract
# --psm 6 : suppose un bloc unique et uniforme de texte (adapté à l'image `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) Originale", "(b) CLAHE + Otsu", "(c) Normalisation fond + Otsu",
            "(d) Normalisation fond (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]
```

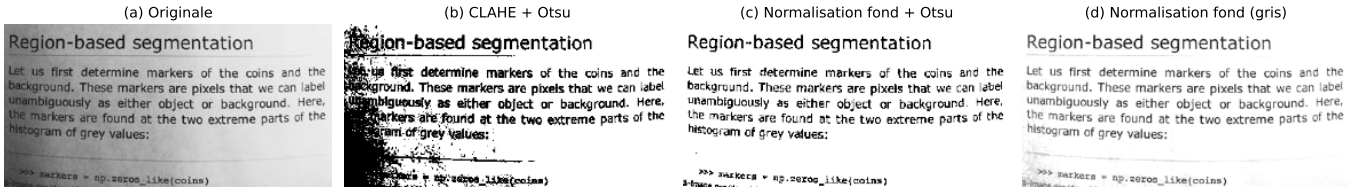


Figure 7.3: Comparaison du texte extrait par Tesseract à partir de quatre versions de la même image : (a) image originale ; (b) CLAHE suivi d'un seuillage par Otsu ; (c) normalisation du fond suivie d'un seuillage par Otsu ; et (d) normalisation du fond en niveaux de gris, sans seuillage. La comparaison montre que la binarisation externe ne favorise pas toujours la reconnaissance, car Tesseract effectue déjà sa propre binarisation adaptative en interne.

Pourquoi le prétraitement améliore-t-il l'OCR ?

Les performances de l'OCR dépendent directement de la qualité de l'image d'entrée. Un faible contraste, un éclairage non uniforme, le bruit et les distorsions géométriques rendent difficile la séparation entre le texte et le fond, augmentant ainsi la probabilité d'erreurs de reconnaissance.
Des techniques telles que la normalisation du fond et l'égalisation adaptative (CLAHE) corrigent les gradients d'éclairage et renforcent le contraste local entre les caractères et le fond, produisant des images plus adaptées à la reconnaissance automatique. La seuillage, quant à lui, doit être appliqué avec prudence : étant une opération irréversible, il peut éliminer les variations subtiles d'intensité sur les bords des caractères — comme l'*anticrénelage* — que le moteur d'OCR lui-même utilise en interne pour résoudre les ambiguïtés entre des symboles visuellement similaires. Pour cette raison, les images en niveaux de gris, corrigées uniquement en ce qui concerne l'éclairage, produisent souvent des transcriptions plus fidèles que leurs versions binarisées.
Dans les documents capturés par des caméras de dispositifs mobiles, le prétraitement tend à offrir un gain de performance plus important que dans les documents numérisés par *scanner*, où l'éclairage est généralement plus uniforme.

7.6 Traduction automatique du texte reconnu

Après la reconnaissance optique de caractères, le texte obtenu peut être soumis à des techniques de traitement du langage naturel, telles que la correction orthographique, l'indexation, la synthèse et la traduction automatique.

La traduction automatique constitue une étape indépendante de l'OCR. Alors que l'OCR convertit les caractères présents dans l'image en texte codé, la traduction opère sur ce texte dans la langue originale du document. De cette manière, les erreurs de reconnaissance peuvent être propagées à la traduction, compromettant la qualité du résultat. Les systèmes actuels de traduction automatique utilisent principalement des architectures neuronales basées sur des mécanismes d'attention et des *transformers* [Bahdanau; Cho; Bengio (2015); Vaswani (2017)].

Dans cet exemple, on utilise le texte obtenu à partir de l'image soumise uniquement à la normalisation du fond, sans seuillage (point **d** de la Figure 7.3), car il présente la transcription la plus fidèle parmi les stratégies évaluées dans la section précédente.

La traduction est réalisée à l'aide de la bibliothèque `deep-translator` (installation : `pip install deep-translator`), qui fournit une interface pour différents services de traduction automatique, y compris Google Traduction.

```
from deep_translator import GoogleTranslator
import shutil as _sh

# Texte obtenu par l'OCR à partir de l'image avec normalisation du fond (tons de gris)
texto_en = texto_norm_gray

if _sh.which("tesseract") is None:
    # Environnement de build sans Tesseract installé (voir cellule précédente) : texte_en
    # est déjà le placeholder pour OCR indisponible, pas du texte réel - sauter la traduction
    # au lieu d'échouer en essayant de traduire une chaîne qui n'est pas vraiment de
    l'anglais.
    texto_pt = "[Tradução indisponível neste ambiente - Tesseract OCR ausente]"
else:
    texto_pt = GoogleTranslator(source="en", target="pt").translate(texto_en)

print("--- Texte original généré par l'image normalisée en tons de gris (OCR, EN) ---")
print(texto_en)

print("\n--- Texte traduit (PT-BR) ---")
print(texto_pt)

mm.show(
    [img_norm],
    titles=["Image avec normalisation du fond (tons de gris)"],
    cols=1,
    figsize=(6, 4)
)
```

```
--- Texte original généré par l'image normalisée en tons de gris (OCR, EN) ---
[Tesseract OCR indisponivel neste ambiente - texto omitido]

--- Texte traduit (PT-BR) ---
[Tradução indisponível neste ambiente - Tesseract OCR ausente]
```

Image avec normalisation du fond (tons de gris)

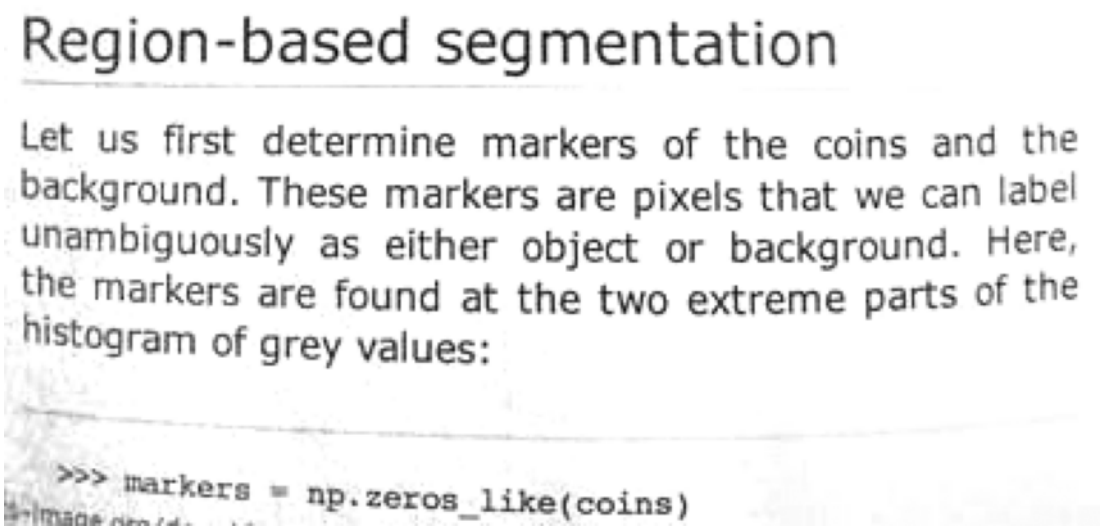


Figure 7.4: Flux de reconnaissance et de traduction automatique. L'image prétraitée par normalisation du fond, sans seuillage, est utilisée comme entrée pour le Tesseract OCR, et le texte reconnu est traduit de l'anglais vers le portugais à l'aide de la bibliothèque *deep-translator*.

> **Pourquoi la qualité de l'OCR influence-t-elle la traduction ?**
>
> La traduction automatique utilise comme entrée le texte produit par l'OCR. Les erreurs de reconnaissance, telles que des caractères incorrects, des mots incomplets ou fragmentés, sont propagées à l'étape de traduction et peuvent modifier le sens du texte.
> Par conséquent, la qualité de la traduction dépend directement de la fidélité de la transcription obtenue par l'OCR. Comme discuté précédemment, cette fidélité n'est pas toujours maximisée par une binarisation externe : des images en niveaux de gris, corrigées uniquement pour l'éclairage, peuvent préserver des informations pertinentes pour la distinction entre des caractères visuellement similaires. Ainsi, le prétraitement de l'image — et le choix adéquat de ses étapes en fonction de la tâche subséquente — contribue à améliorer non seulement la reconnaissance des caractères, mais aussi les performances des étapes ultérieures de traitement du langage naturel, telles que la traduction, l'indexation et la synthèse.

7.7 Fondements de l'OMR et de l'Inspection Industrielle

La **Reconnaissance Optique de Marques** (*Optical Mark Recognition* — OMR) est une technique de Vision par Ordinateur destinée à l'identification automatique de marques à des positions préalablement définies sur un formulaire. Ses applications incluent les feuilles de réponses, les questionnaires, les formulaires administratifs et autres documents structurés.

Contrairement à l'OCR (*Optical Character Recognition*), qui reconnaît des caractères et des mots, l'OMR détermine la présence, l'absence ou l'intensité de marques dans des régions préalablement connues. Au lieu d'interpréter du texte, elle exploite des propriétés géométriques et statistiques associées au remplissage de ces régions.

Les systèmes modernes d'OMR traitent des images obtenues par des *scanners*, des caméras ou des dispositifs mobiles, automatisant des tâches qui dépendaient auparavant d'équipements spécialisés.

De manière générale, un système d'OMR comprend les étapes suivantes :

1. **Acquisition :** conversion du document physique au format numérique ;
2. **Prétraitement :** correction géométrique, réduction du bruit et binarisation ;

3. **Localisation des régions d'intérêt :** identification des zones destinées aux marques ;
4. **Analyse des marques :** évaluation du remplissage des régions candidates ;
5. **Interprétation :** conversion des marques en réponses ou en données structurées.

Ces principes s'étendent naturellement à l'**Inspection Industrielle Automatisée**. Sur les chaînes de production, le même enchaînement — acquisition, prétraitement, segmentation, extraction de caractéristiques et décision — est utilisé pour détecter des défauts de surface, vérifier l'intégrité des composants et mesurer des dimensions avec une précision subpixel. La différence réside dans le domaine d'application : tandis que l'OMR opère sur des documents à structure prédéfinie, l'inspection industrielle traite des objets dont les variations géométriques et radiométriques doivent être modélisées de manière plus flexible.

Dans les sections suivantes, ces deux applications sont développées au moyen de projets pratiques qui reproduisent des étapes typiques de systèmes réels.

7.8 Projets Pratiques : Construction d'un *Pipeline* d'Analyse Documentaire

Les concepts de ce chapitre seront développés à travers des projets qui reproduisent des étapes typiques de systèmes réels d'analyse documentaire, en introduisant des techniques réutilisables dans des applications d'OCR, d'OMR, d'inspection visuelle et de traitement de formulaires.

7.8.1 Alignement Automatique de Documents (*Prétraitement OCR/OMR*)

La correction d'inclinaison (*deskew*) est une étape fondamentale dans le traitement des documents. Les rotations introduites lors de la numérisation ou de la capture compromettent la localisation des régions d'intérêt et réduisent la précision des étapes ultérieures.

Dans ce projet, un système sera développé pour estimer automatiquement l'orientation prédominante du document et corriger son inclinaison. Pour cela, des techniques classiques de détection de contours avec l'opérateur de Canny et la détection de droites par la Transformée de Hough seront employées. À partir des lignes identifiées, l'angle de rotation sera estimé et une transformation affine sera appliquée pour produire une version alignée du document.

Comme les formulaires et les feuilles de réponses sont fréquemment distribués au format PDF, le *pipeline* commence par la rastérisation de chaque page, la convertissant en une image matricielle. Dans ce chapitre, cette étape sera réalisée avec la bibliothèque `pdf2image`, générant des images PNG avec une résolution de 300 DPI (*dots per inch*). À partir de celles-ci, les techniques de détection de contours, de Transformée de Hough, de segmentation, d'extraction de contours et de reconnaissance automatique de motifs étudiées tout au long du chapitre pourront être appliquées.

```
import os
import urllib.request
from pdf2image import convert_from_path
from skimage import data
import cv2

# Répertoire des microdonnées et feuilles de réponses de l'examen institutionnel
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 le fichier n'existe pas localement, le télécharge automatiquement depuis GitHub
if not os.path.exists(file_path):
    print(f"[TÉLÉCHARGEMENT] Téléchargement du PDF depuis GitHub : {url_github}")
    try:
```

```
        # Garantit que le dossier 'données' existe avant de l'enregistrer
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
        urllib.request.urlretrieve(url_github, file_path)
        print("[TÉLÉCHARGEMENT] PDF téléchargé avec succès !")
    except Exception as e:
        print(f"[TÉLÉCHARGEMENT] Échec du téléchargement du fichier : {e}")

print(f"PDF de feuilles d'examen numérisées : {file_path}")

if os.path.exists(file_path):
    # Rasterisation des pages avec une résolution optimisée 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"[INGESTION] Page PDF convertie avec succès : {saida}")
else:
    print("[AVERTISSEMENT] Fichier PDF introuvable dans le chemin. Activation du repli via
    skimage.data.")
    # Injecte une matrice de texte publique pour garantir l'exécution continue du pipeline
    img_fallback = data.text()
    cv2.imwrite("test02.png", img_fallback)
    print("[INGESTION] Image de repli structurée : test02.png")

# Charge et affiche l'image rasterisée initiale à l'aide de l'écosystème morph
if os.path.exists('test02.png'):
    img_original = mm.read('test02.png')
else:
    # Repli définitif au cas où même skimage échoue
    img_original = np.ones((400, 400), dtype=np.uint8) * 255

mm.show(img_original, figsize=(4, 3))
```

```
PDF de feuilles d'examen numérisées : dados/provas_qrcode_EP.pdf
[INGESTION] Page PDF convertie avec succès : test01.png
[INGESTION] Page PDF convertie avec succès : test02.png
[INGESTION] Page PDF convertie avec succès : test03.png
```

Figure 7.5: *Pipeline* d'ingestion de documents : rasterisation adaptative de pages PDF en matrices discrètes au format PNG, affichant la page deux.

7.8.2 Algorithme de redressement d'inclinaison (*Deskew*)

L'étape de *deskew* vise à estimer et corriger l'inclinaison globale d'un document numérisé, en alignant son contenu sur les axes de l'image. La Figure 7.7 présente le flux complet de traitement, depuis l'image originale jusqu'au résultat après la correction géométrique. En complément, le simulateur de la Figure 7.6 permet de visualiser le fonctionnement de la Transformée de Hough et de comprendre comment l'orientation prédominante est estimée.

La procédure se compose de trois étapes principales :

1. détection des contours par l'opérateur de Canny ;
2. estimation de l'orientation prédominante au moyen de la Transformée de Hough Linéaire ;
3. correction de l'inclinaison à l'aide d'une transformation affine de rotation.

Après le redressement, le document présente une orientation approximativement horizontale, ce qui favorise les étapes ultérieures de segmentation, d'étiquetage des composantes connexes et de reconnaissance des caractères et des marques.

7.8.3 Modélisation Mathématique

Les sous-sections suivantes formalisent, en termes mathématiques, les étapes décrites précédemment, en reliant le gradient de l'image, la paramétrisation des droites dans l'espace de Hough et la matrice de rotation utilisée pour la correction géométrique.

7.8.3.1 Détection des contours

Initialement, l'image est lissée par un filtre gaussien, réduisant l'effet des bruits haute fréquence susceptibles de générer des contours parasites. Les concepts de filtrage spatial et de convolution ont été présentés dans le **Chapitre 3**.

Ensuite, l'opérateur de Canny estime le gradient de l'image. Soit $f(x, y)$ l'intensité de l'image et α l'angle de rotation.

La magnitude du gradient est donnée par

$$|\nabla f(x, y)| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}.$$

où :

- $f(x, y)$ représente l'intensité de l'image à la position (x, y) ;
- $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial y}$ sont les dérivées partielles dans les directions horizontale et verticale ;
- $|\nabla f(x, y)|$ est la magnitude du gradient.

Après le calcul du gradient, l'algorithme applique la suppression des non-maxima (*non-maximum suppression*) et le seuillage par hystérésis, produisant une image binaire contenant les principaux contours du document.

7.8.3.2 Transformée de Hough

L'image binaire des contours est traitée par la Transformée de Hough linéaire, dont l'objectif est de détecter les structures approximativement rectilignes. Au lieu de la représentation cartésienne de la droite, $y = ax + b$, on utilise la représentation sous forme normale,

$$\rho = x \cos\theta + y \sin\theta,$$

où :

- x et y sont les coordonnées d'un point appartenant à la droite ;

- ρ est la distance perpendiculaire entre la droite et l'origine du système de coordonnées de l'image ;
- θ est l'angle formé entre la normale à la droite et l'axe horizontal de l'image.

Dans cette représentation, chaque point de contour (x, y) génère une courbe dans l'espace des paramètres (ρ, θ). L'intersection des courbes produites par les points appartenant à une même droite donne naissance à des maxima dans une matrice bidimensionnelle appelée **accumulateur**. Ainsi, les pics de l'accumulateur correspondent aux droites prédominantes de l'image, telles que les bords du document, les lignes de formulaires ou les lignes de texte.

Pour estimer l'inclinaison globale du document, on ne considère que les droites dont les angles satisfont

$$-45^\circ \leq \theta \leq 45^\circ.$$

Cette restriction élimine les orientations incompatibles avec la disposition attendue du document et réduit l'influence des droites verticales ou des structures non pertinentes. Soit $\theta_1, \theta_2, \ldots, \theta_n$ l'ensemble des angles des droites sélectionnées. L'estimation de l'inclinaison globale est obtenue par la médiane,

$$\hat{\theta} = \operatorname{med}(\theta_1, \theta_2, \ldots, \theta_n),$$

où :

- θ_i est l'angle de la i-ième droite détectée par la Transformée de Hough ;
- n est le nombre de droites considérées après le filtrage angulaire ;
- $\hat{\theta}$ est l'estimation de l'inclinaison globale du document.

La médiane est adoptée car elle est moins sensible à la présence de détections isolées (*outliers*) que la moyenne arithmétique, produisant une estimation plus stable de l'orientation prédominante.

7.8.3.3 Rotação affine

Soit $\hat{\theta}$ l'inclinaison estimée à l'étape précédente. La correction géométrique consiste à appliquer une transformation affine de rotation autour du centre de l'image, de sorte que l'orientation prédominante coïncide avec l'axe horizontal. Les transformations affines ont été étudiées au **chapitre 2**, ainsi que les opérations de translation, de mise à l'échelle, de cisaillement et de rotation.

Soient (x, y) la position d'un pixel par rapport au centre de l'image et (x', y') sa position après la rotation. La transformation est décrite par

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = R(\alpha) \begin{bmatrix} x \\ y \end{bmatrix},$$

où

$$R(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix},$$

avec :

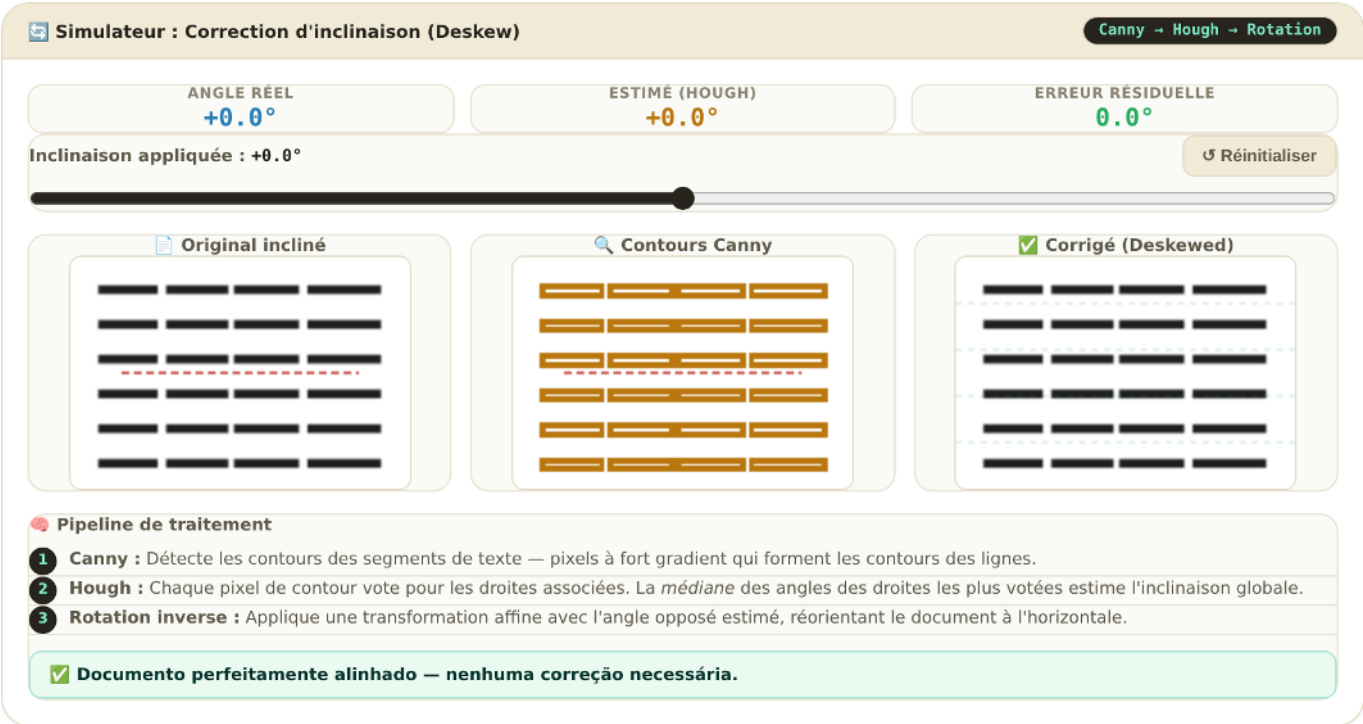
- (x, y) les coordonnées originales du pixel par rapport au centre de l'image ;
- (x', y') les coordonnées du pixel après la rotation ;
- α l'angle de rotation appliqué pour compenser l'inclinaison estimée du document ;
- $R(\alpha)$ la matrice de rotation.

En pratique, l'angle appliqué correspond à l'opposé de l'inclinaison estimée,

$$\alpha = -\hat{\theta},$$

où $\hat{\theta}$ représente l'orientation prédominante obtenue par la transformée de Hough.

Comme les coordonnées transformées ne coïncident pas toujours avec des positions entières de la grille de pixels, il est nécessaire de rééchantillonner l'image pour déterminer les nouvelles valeurs d'intensité. Dans l'implémentation présentée dans ce chapitre, la fonction `mm.rotate` effectue cette opération en utilisant une interpolation bicubique, réduisant ainsi les artefacts de rééchantillonnage et préservant la continuité visuelle des bords et des caractères.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap06/sim-06-deskew.html

Figure 7.6: Simulateur interactif de correction d'inclinaison (*deskew*) : déplacez le *curseur* pour incliner le document et observez les trois étapes du *pipeline* — image inclinée, contours Canny et résultat corrigé.

```
def retificar_inclinacao_documento(img):

    gray = mm.gray(img) if img.ndim == 3 else img
    edges = cv2.Canny(cv2.GaussianBlur(gray, (5,5), 0), 50, 150)
    lines = cv2.HoughLines(edges, 1, np.pi/180, 200)

    if lines is None:
        return edges, img

    angulos = []
    for line in lines:
        angulo = np.rad2deg(line[0][1]) - 90
        if -45 < angulo < 45:
            angulos.append(angulo)

    if not angulos:
        return edges, img

    return edges, mm.rotate(img, np.median(angulos), interp="bicubic")

# Exécution du pipeline de deskew
img_edges, img_final = retificar_inclinacao_documento(img_original)

# Affichage multiple standardisé avec le format natif du livre
mm.show(
```

```
    [img_original, img_edges, img_final],
    titles=["Image d'origine", "Contours de Canny", "Document redressé"],
    cols=3,
    figsize=(12, 4)
)
```

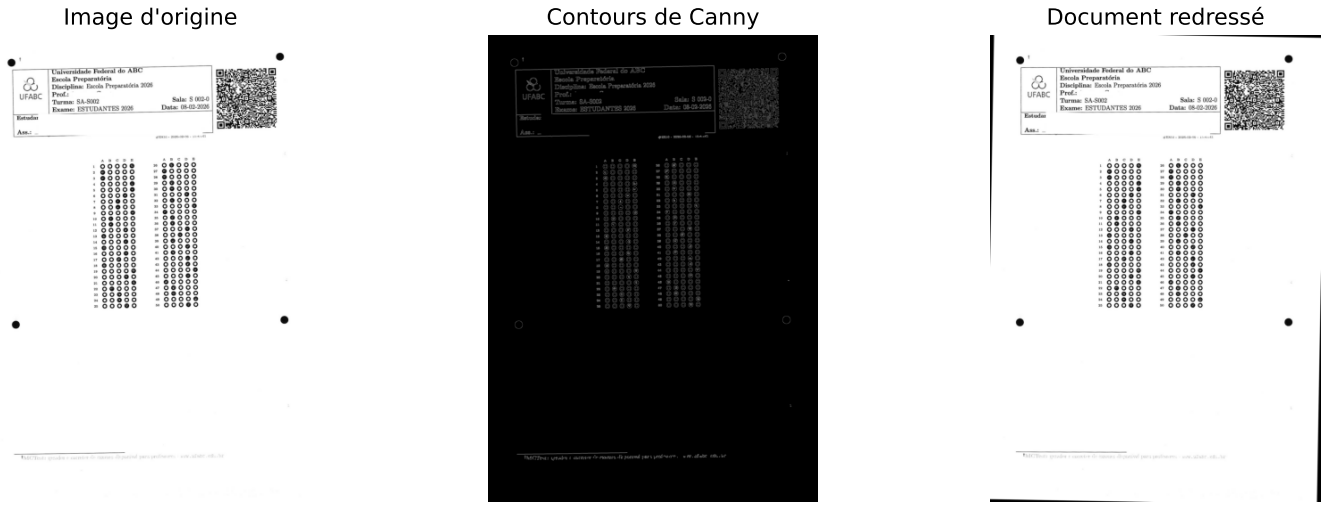


Figure 7.7: *Pipeline* de redressement axial : affichage comparatif entre l'entrée rotée d'origine, la carte de gradients structurels de Canny et le résultat final aligné avec un fond normalisé en blanc.

> **Pourquoi cela fonctionne-t-il ? — Transformée de Hough**
>
> Dans la transformée de Hough, chaque pixel de bord contribue avec des votes pour toutes les droites qui peuvent passer par sa position. Au lieu de sélectionner uniquement la droite ayant le plus grand nombre de votes, l'algorithme considère toutes les droites dont la quantité de votes dépasse un seuil minimal et calcule leurs angles respectifs. L'inclinaison globale du document est alors estimée par la médiane de ces angles, une mesure robuste aux valeurs aberrantes. Ainsi, les droites parasites produites par les ombres, les bruits ou d'autres éléments de l'image exercent peu d'influence sur l'estimation finale, tant que la majorité des droites détectées correspond aux bords du document.

7.8.4 Limites pratiques

Bien qu'il présente de bonnes performances dans des conditions de numérisation usuelles, cette méthode dépend de l'existence de structures linéaires suffisamment définies pour être détectées par la transformée de Hough, telles que les bords de page, les lignes de formulaires ou les lignes de texte. Sa précision peut être réduite dans des images à faible résolution, avec un bruit excessif, des ombres intenses ou de grandes inclinaisons. En général, les documents numérisés avec une résolution proche de 300 DPI et un éclairage homogène fournissent des résultats adéquats pour des applications d'OCR et d'OMR.

L'implémentation présentée dans ce chapitre a une finalité didactique, illustrant les principes de la correction automatique de l'inclinaison par la détection des contours, la transformée de Hough et la rotation affine. En utilisant uniquement l'orientation des structures linéaires prédominantes, cette méthode peut être appliquée à différents types de documents, sans dépendre de marqueurs spécifiques.

Dans les systèmes réels d'analyse documentaire, cependant, l'alignement utilise généralement des marqueurs géométriques préalablement connus. Dans le modèle de feuille de réponses employé par l'écosystème MCTest, par exemple, quatre disques noirs de référence sont utilisés, en plus des régions correspondant à l'en-tête, au *QRCode* et aux cadres de réponses. La localisation de ces éléments permet d'estimer simultanément la rotation, l'échelle et la translation de la feuille, rendant le recalage moins sensible à la quantité de texte, à l'absence de lignes structurelles et aux variations d'impression ou de numérisation.

Pour cette raison, l'approche basée sur la transformée de Hough est utilisée dans ce chapitre pour introduire les fondements du problème, tandis que les étapes ultérieures adoptent l'alignement par marqueurs géométriques, stratégie prédominante dans les systèmes d'OMR et d'analyse documentaire.

7.8.5 Détection des bords et des contours

La localisation précise des régions d'intérêt est une étape essentielle dans les systèmes d'OMR. Dans le modèle de feuille de réponses utilisé dans ce chapitre, l'en-tête et le cadre de réponses sont contenus dans un rectangle virtuel délimité par quatre disques noirs positionnés aux coins. L'identification de ces marqueurs permet de localiser la région d'intérêt et de corriger les distorsions géométriques introduites lors de l'acquisition de l'image.

La procédure comprend cinq étapes. Initialement, on applique une **fermeture morphologique** (dilatation suivie d'une érosion), opération étudiée au **Chapitre 4**, en utilisant un élément structurant en forme de disque (`mm.sedisk(33)`). Cette opération réduit les petites discontinuités et préserve les disques de référence, les rendant plus homogènes. Ensuite, l'image est inversée (`mm.neg`), de sorte que les disques constituent désormais des composantes claires sur fond sombre.

À l'étape suivante, on applique l'opération `mm.edgeoff` (**Chapitre 4**), qui supprime les composantes connectées aux bords de l'image, éliminant les artefacts tels que les ombres de numérisation, les marques de coupe et autres objets parasites sur les marges. Les composantes restantes sont ensuite analysées à partir de leurs contours et filtrées selon des propriétés géométriques, comme l'aire et la circularité, afin d'identifier les disques candidats. L'implémentation du MCTest rend ce processus plus robuste en sélectionnant, parmi tous les candidats, les quatre dont les centres forment un rectangle avec une largeur compatible avec celle de l'image, réduisant ainsi l'occurrence de faux positifs.

Enfin, les centres des quatre disques sont ordonnés spatialement (en haut à gauche, en haut à droite, en bas à gauche et en bas à droite) et utilisés comme points de contrôle dans une transformation de perspective (*perspective warp*). Cette transformation redresse l'image, produisant une représentation alignée et avec des dimensions connues, adaptée aux étapes subséquentes de segmentation et de reconnaissance.

Les principales étapes de ce *pipeline*, depuis le traitement morphologique jusqu'à l'image redressée, sont illustrées ci-dessous.

```
import cv2
import numpy as np
from morph import mm

# img: image en niveaux de gris de la feuille de réponse
if img_final.ndim == 2:
    img = img_final
else:
    img = mm.gray(img_final)

# 1. Fermeture morphologique : préserve les disques sombres, en supprimant tout ce qui est
plus petit que le disque
img_close = mm.close(img, mm.sedisk(41))

# 2. Inversion : les disques sombres deviennent des composants clairs sur fond sombre
img_neg = mm.neg(img_close)

# 3. Supprime les composants connectés qui touchent le bord de l'image
img_edgeoff = mm.edgeoff(img_neg)

# 4. Extraction des contours externes
contornos, _ = cv2.findContours(img_edgeoff, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# 5. Filtrage par aire et circularité, en ne conservant que les 4 disques
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))

# Vérification robuste : interrompt le pipeline avec un message clair au lieu d'une
AssertionError
if len(centros) != 4:
    print(f"[AVERTISSEMENT] 4 disques marqueurs attendus, trouvé {len(centros)}.")
    print("  Vérifiez que l'image est une feuille de réponses MCTest valide")
    print("  ou ajustez les paramètres de circularité et d'aire minimale.")
    img_retificada = img  # repli : préserve l'image sans redressement
else:
    # 6. Tri des centres : supérieur-gauche, supérieur-droit,
    # inférieur-gauche, inférieur-droit
    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. Redressement par transformation de perspective (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=["Fermeture (sedisk 41)", "edgeoff", "Redressée (warp)"],
        cols=3,
        figsize=(12, 4)
    )

mm.write(img_retificada, "img_beetween_disks.png")
```

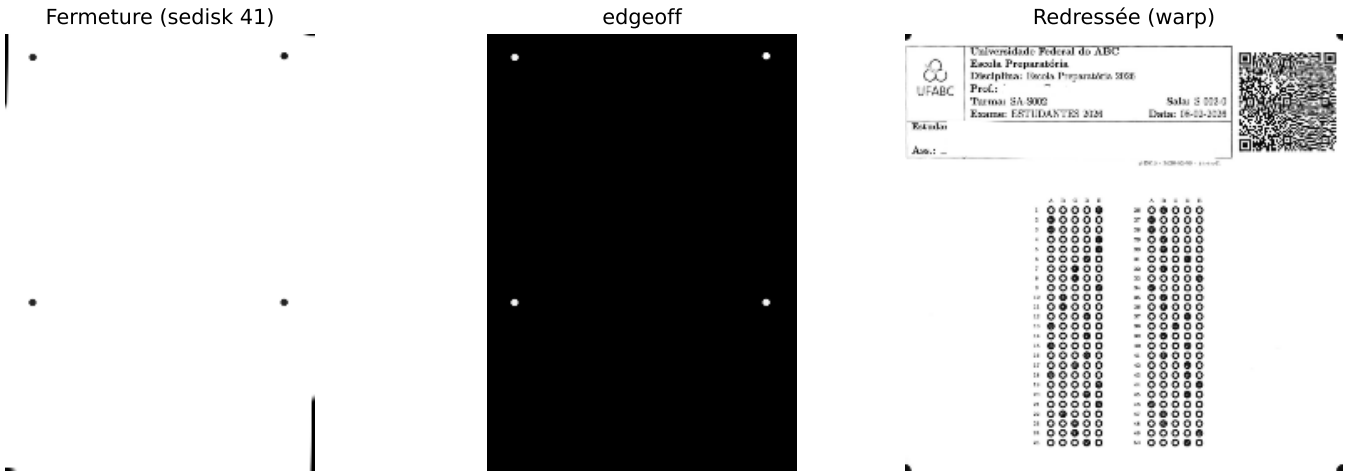


Figure 7.8: Détection des disques marqueurs, extraction des contours et redressement par transformation de perspective.

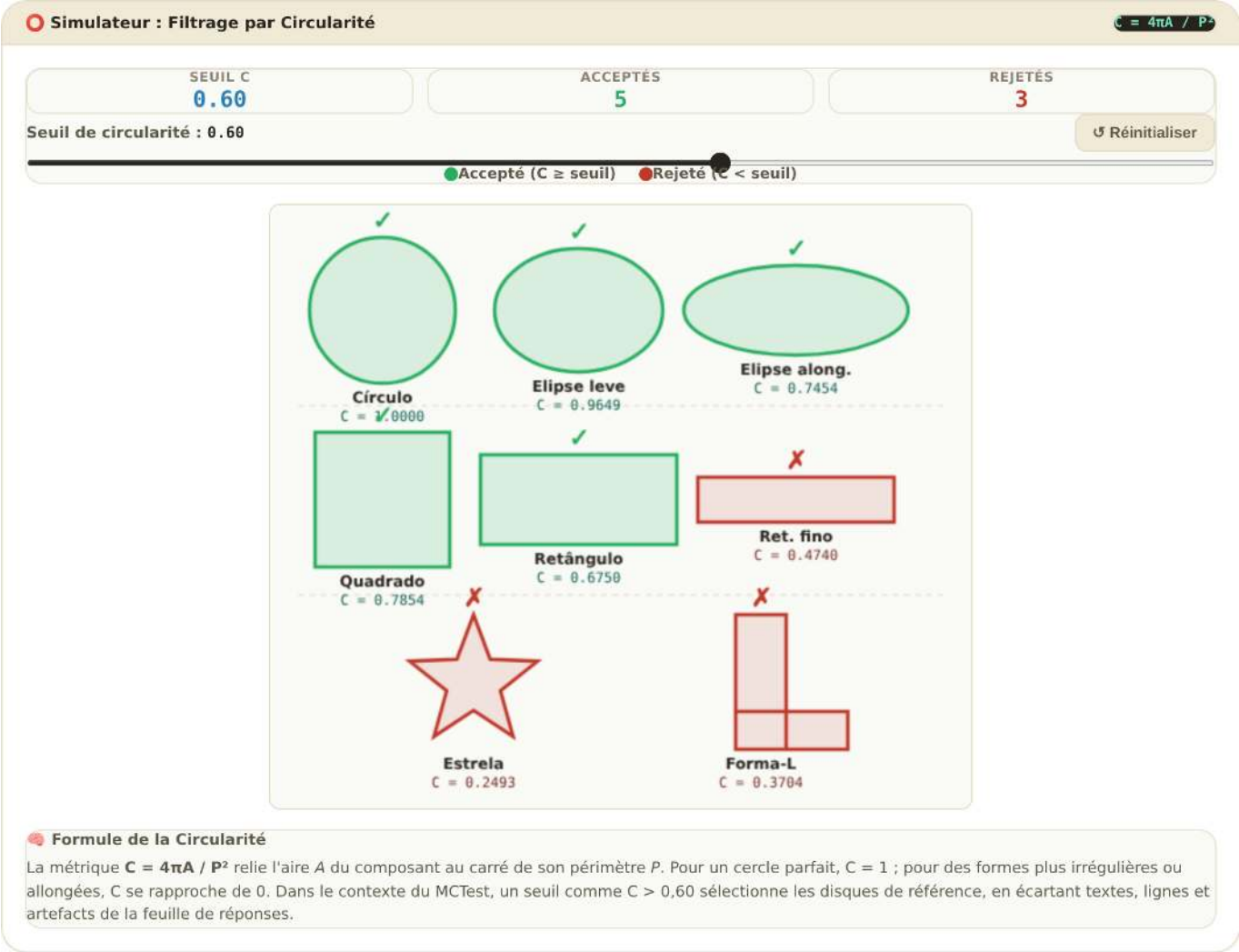
> 🧠 Pourquoi cela fonctionne-t-il ? — De la fermeture morphologique à la rectification
>
> **Fermeture morphologique :** la dilatation suivie de l'érosion comble les petites discontinuités et adoucit les contours des objets sans modifier significativement leur forme globale. En utilisant un élément structurant de grande taille (`sedisk(41)`), les détails fins, tels que les textes, les lignes du formulaire et les petits bruits, tendent à être intégrés au fond pendant le traitement, tandis que les objets de plus grande échelle, comme les disques de référence, restent préservés et deviennent plus homogènes.
> **Circularité :** après l'isolement des composants candidats, la métrique $C = \frac{4\pi A}{P^2}$ quantifie à quel point leur forme se rapproche d'un cercle. Sa valeur est égale à 1 pour un cercle parfait et diminue à mesure que le contour devient plus irrégulier. Ainsi, un seuil tel que $C > 0{,}8$ permet d'éliminer la plupart des faux positifs sans recourir à des modèles d'apprentissage. Un simulateur de cette métrique est présenté dans la Figure 7.9.
> **Transformation de perspective (*perspective warp*) :** une fois les quatre disques de référence identifiés, leurs centres sont utilisés comme points de contrôle pour estimer la transformation projective qui mappe l'image capturée au plan du document. Cette transformation corrige les distorsions introduites par la perspective lors de l'acquisition de l'image, produisant une représentation frontale aux dimensions connues et adaptée aux étapes ultérieures de segmentation et de reconnaissance.

7.8.6 Isollement, Segmentation et Décodage du *QRCode*

Après la rectification géométrique de la feuille de réponses, on procède à la détection et au décodage du *QRCode* présent dans le formulaire. Ce marqueur stocke des informations utilisées par le système d'OMR (*Optical Mark Recognition*), telles que l'identification de l'étudiant, le code de l'épreuve et sa variante, permettant la récupération du corrigé correspondant dans la base de données. Par sécurité, ces informations sont chiffrées avant la génération du *QRCode*. Ainsi, la séquence décodée correspond à une *chaîne* hexadécimale, dont l'interprétation est réalisée exclusivement par le système MCTest. La procédure se compose de trois étapes : le prétraitement morphologique, l'isolement de la région du *QRCode* et le décodage de son contenu.

Initialement, l'image rectifiée en niveaux de gris est binarisée au moyen de l'opération `mm.threshold`. Ensuite, on applique une **ouverture morphologique** (érosion suivie d'une dilatation), étudiée au **Chapitre 4**, à l'aide d'un élément structurant carré (`mm.sebox(2)`). Cette opération élimine les petits bruits et lisse les imperfections sans compromettre la structure du marqueur. Enfin, l'image est inversée (`mm.neg`), de sorte que le *QRCode* constitue désormais un composant clair sur fond sombre, facilitant l'extraction de ses contours.

La localisation du *QRCode* est réalisée par l'analyse des contours externes de l'image binarisée. Parmi les composants détectés, on sélectionne celui présentant la plus grande aire et une géométrie approximativement



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap06/sim-06-circularidade.html

Figure 7.9: Simulateur interactif de filtrage par circularité : déplacez le *curseur* pour ajuster le seuil C et observez quels composants sont acceptés (vert) ou rejetés (rouge).

carrée, en écartant les autres éléments imprimés de la feuille. Ensuite, la région correspondante est élargie d'une petite marge de sécurité, garantissant la préservation intégrale du marqueur.

Le *QRCode* est alors extrait directement de l'image rectifiée en niveaux de gris, préservant sa qualité radiométrique. Comme cette région présente généralement des dimensions réduites, on applique un redimensionnement avec interpolation cubique, augmentant la résolution spatiale et favorisant l'identification de ses modules. La lecture est effectuée par le détecteur de *QRCode* d'OpenCV (`cv2.QRCodeDetector`), qui récupère la séquence de caractères originellement codée.

Le flux complet de ce traitement, depuis le prétraitement morphologique jusqu'au décodage du *QRCode*, est illustré dans la Figure 7.10. L'extrait affiché en sortie correspond uniquement au début de la *chaîne* hexadécimale chiffrée ; son interprétation complète est réalisée en interne par MCTest après le décodage.

```
import cv2
import numpy as np
from morph import mm

# f : image redressée convertie au bon type 8 bits (0-255)
f = img_retificada.astype('uint8')

# 1. Seuillage : conversion de l'image en niveaux de gris en image binaire
f_thresh = mm.threshold(f)

# 2. Ouverture morphologique : élimine les petits bruits et adoucit le contour des blocs
f_open = mm.open(f_thresh, mm.sebox(2))

# 3. Inversion morphologique : les modules sombres deviennent des composants clairs sur fond
sombre
f_inv = mm.neg(f_open)

# 4. Conversion sûre en uint8 avec échelle 0-255
img_uint8 = (f_inv.astype(np.uint8) * 255) if f_inv.max() == 1 else f_inv.astype(np.uint8)

# 5. Détection des contours externes
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. Filtrage par le plus grand contour avec proportion approximativement carrée
#    (ratio d'aspect entre 0,7 et 1,3 écarte les rectangles allongés de la feuille)
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."
    )

# Sélectionne le plus grand candidat carré par aire de boîte englobante
maior_contorno = max(candidatos, key=lambda c: cv2.boundingRect(c)[2] *
cv2.boundingRect(c)[3])
x, y, w, h = cv2.boundingRect(maior_contorno)

# 7. Expansion de la boîte englobante avec marge de sécurité (évite le tronquage du QRCode)
margem = 5
```

```python
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. Recadrage de la région d'intérêt à partir de l'image originale (nette, en gris)
img_qrcode_final = img_retificada[y1:y2, x1:x2]

# 9. Agrandissement à la résolution minimale de décodage (400 px sur le côté le plus long)
#    cv2.QRCodeDetector requiert des modules d'au moins 3-4 px de largeur pour décoder
#    en toute sécurité ; les images plus petites que ~400 px ont tendance à échouer.
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. Initialisation du détecteur natif de QRCode d'OpenCV
detector = cv2.QRCodeDetector()

# 11. Détection géométrique et décodage des données textuelles
dados, pontos, qrcode_reto = detector.detectAndDecode(img_para_leitura)


# Visualisation intermédiaire : progression de la binarisation à l'isolation du QRCode
mm.show(
    [f_thresh, f_open, f_inv, img_qrcode_final],
    titles=["1. Seuillage", "2. Ouverture morphologique", "3. Inversion", "Image finale"],
    cols=3, figsize=(12, 4)
)

# Validation et sortie des métadonnées extraites
if dados:
    print(f"QRCode décodé avec succès : \n{dados[:50]}...")
else:
    raise ValueError(
        "Falha na decodificação do QRCode. "
        "Verifique o limiar, as margens da região ou a qualidade da imagem."
    )
```

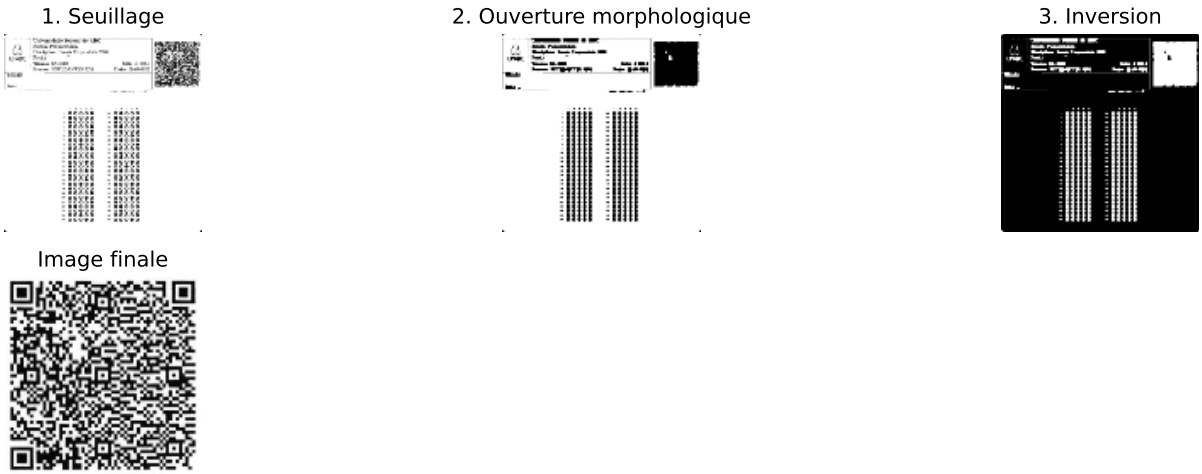


Figure 7.10: *Pipeline* de traitement du *QRCode* : seuillage, ouverture morphologique, inversion et recadrage final pour décodage.

```
QRCode décodé avec succès :
325a356b71367266556955646b7233454149624a694f417730...
```

Pourquoi cela fonctionne-t-il ? — Isolement et décodage du *QRCode*

Ouverture morphologique : contrairement à la fermeture, l'ouverture (érosion suivie d'une dilatation) élimine les petits bruits et protubérances sans modifier significativement la géométrie des objets plus grands. Ainsi, elle préserve la structure du *QRCode* tout en supprimant les composants parasites qui pourraient entraver sa localisation.
Sélection par géométrie : le *QRCode* possède une forme approximativement carrée ($w/h \approx 1$). La combinaison de ce critère avec la sélection du composant de plus grande aire écarte les lignes du formulaire, les textes et autres éléments imprimés, permettant d'isoler le marqueur sans recourir à des modèles d'apprentissage.
Redimensionnement avant décodage : lorsque le *QRCode* occupe peu de pixels dans l'image, ses modules deviennent difficiles à distinguer. Le redimensionnement avec interpolation cubique augmente la résolution spatiale de la région d'intérêt, facilitant l'identification des motifs du code par le `cv2.QRCodeDetector` et rendant le décodage plus robuste.

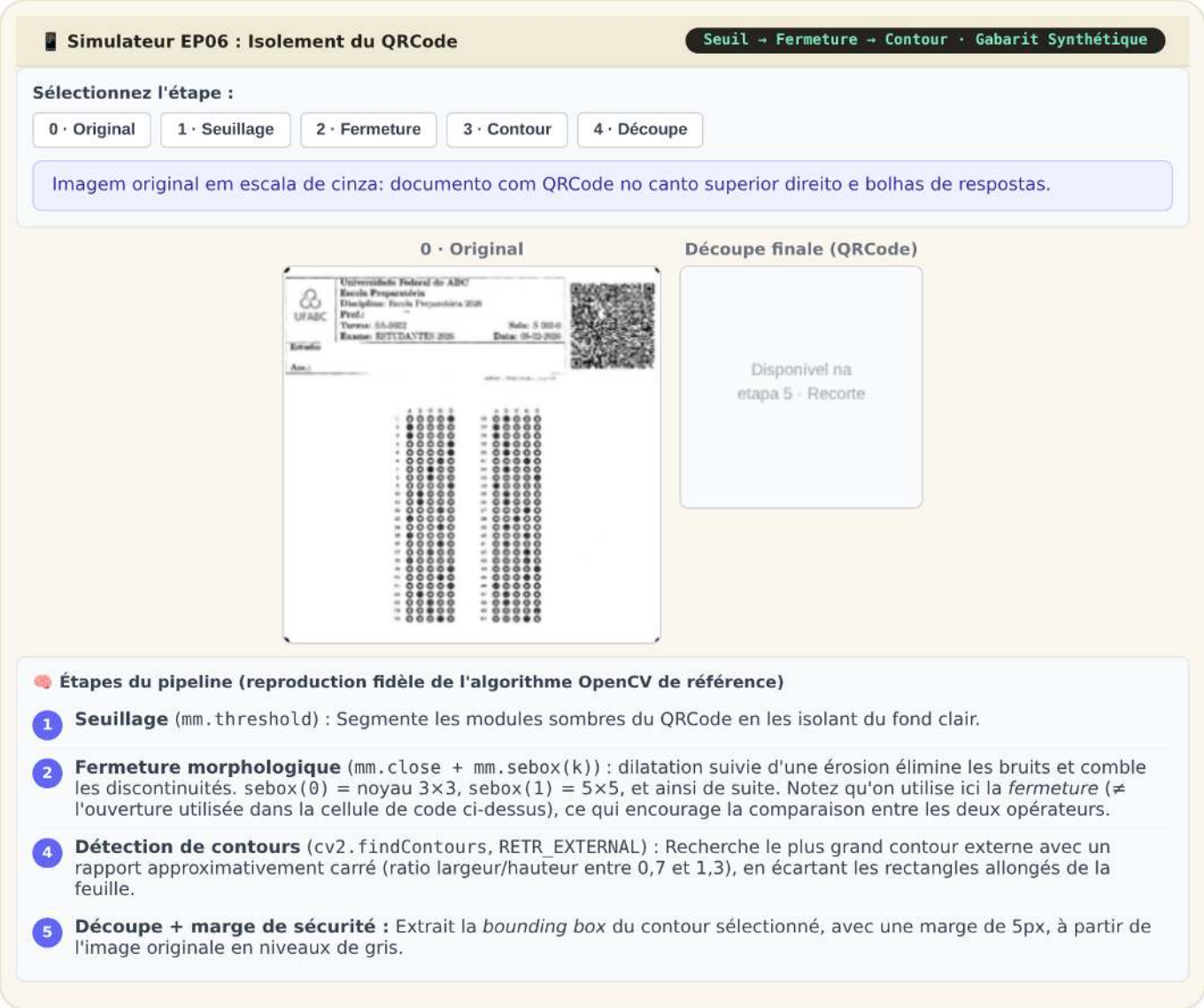
7.8.7 Décodage de codes-barres

Dans la section précédente, le décodage des *QRCodes* a été réalisé à l'aide du détecteur natif d'OpenCV (`cv2.QRCodeDetector`), qui intègre dans une interface unique la détection géométrique du symbole, sa rectification et l'extraction de l'information codée.

Pour les codes-barres linéaires (1D), tels que EAN-13, Code 39 et Code 128, une alternative largement utilisée est la bibliothèque `pyzbar`. Contrairement au `QRCodeDetector`, elle prend en charge diverses symbologies de codes-barres et peut également être employée pour la lecture de *QRCodes*.

La procédure consiste à localiser automatiquement chaque symbole présent dans l'image et à interpréter la séquence de barres et d'espaces correspondante, produisant ainsi la chaîne de caractères codée. Outre les données décodées, la bibliothèque fournit des informations telles que la symbologie identifiée et la position du code dans l'image, permettant sa validation ou son traitement ultérieur.

La Figure 7.12 présente un exemple de code-barres et le résultat de son décodage à l'aide de la bibliothèque `pyzbar`.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap06/sim-06-qrcode.html

Figure 7.11: Simulateur interactif du *pipeline* d'isolation du QRCode : parcourez les étapes de filtrage, ajustez l'élément structurant de la fermeture morphologique et voyez la détection du contour carré sur l'image originale du gabarit.

```python
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 le fichier n'existe pas localement, télécharge automatiquement depuis GitHub
if not os.path.exists(barcode_path):
    print(f"[TÉLÉCHARGEMENT] Téléchargement du code-barres depuis GitHub : {url_github}")
    try:
        os.makedirs(os.path.dirname(barcode_path), exist_ok=True)
        urllib.request.urlretrieve(url_github, barcode_path)
        print("[TÉLÉCHARGEMENT] Image téléchargée avec succès !")
    except Exception as e:
        print(f"[TÉLÉCHARGEMENT] Échec du téléchargement du fichier : {e}")

if os.path.exists(barcode_path):
    image = mm.read(barcode_path)
else:
    # Repli synthétique : génère un motif de barres verticales simulant un Code-128
    print("[AVERTISSEMENT] Fichier 'données/barcode.png' introuvable.")
    print("        Utilisation d'une image synthétique pour la démonstration du pipeline.")
    h, w = 100, 400
    img_synth = np.ones((h, w), dtype=np.uint8) * 255
    # Barres sombres à positions régulières (motif simplifié)
    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)

# Exécute le décodage avec pyzbar
try:
    barcodes = decode(image)
except ImportError:
    print("[ERREUR] zbar du système introuvable. Exécutez : !apt-get install -y libzbar0")
    barcodes = []

if barcodes:
    dados_bc = barcodes[0].data.decode("utf-8")
    tipo = barcodes[0].type
    print(f"Code-barres décodé avec succès [{tipo}]:\n{dados_bc}")
else:
    print("[INFO] Aucun code-barres détecté dans l'image.")
    print(
        "Sur une image synthétique, c'est attendu - "
        "remplacez par le fichier réel pour décoder."
    )
```

Figure 7.12: Décodage de code-barres linéaire avec *pyzbar* : image d'entrée et données extraites.

```
Code-barres décodé avec succès [EAN13]:
0000000000055
```

7.9 Le MCTest comme étude de cas : du prototype au système en production

Jusqu'à présent, les principales étapes du *pipeline* de traitement ont été présentées et analysées individuellement, notamment la correction de l'inclinaison (*deskew*), la détection des marqueurs, la rectification par perspective et la lecture des *QRcodes*. Bien que cette approche facilite la compréhension de chaque technique, les applications réelles exigent l'intégration de ces étapes dans un flux de traitement unique et cohérent.

Le **MCTest** constitue un exemple de cette intégration. Développé à l'UFABC et disponible en tant que logiciel open source, le système est utilisé depuis 2012 pour la correction automatisée d'évaluations, offrant un support pour différents modèles de feuilles de réponses, des corrigés individualisés et la génération automatique de rapports de performance (ZAMPIROLLI, 2023).

À partir de ce point, l'accent n'est plus mis sur l'implémentation isolée des algorithmes, mais sur l'organisation de ces algorithmes dans une application complète. Outre la qualité des méthodes de traitement d'images, un système de cette nature doit répondre à des exigences telles que la robustesse face à différentes conditions d'acquisition, la facilité de maintenance et la capacité d'évolution vers de nouvelles fonctionnalités.

Dans les sections suivantes, le module de Vision par Ordinateur du MCTest, implémenté dans le fichier `CVMCTest.py`, sera analysé. L'objectif est de montrer comment les concepts présentés tout au long de ce chapitre sont combinés dans un *pipeline* de traitement utilisé dans une application réelle.

7.9.1 Obtention et préparation du module

Le fichier `CVMCTest.py` intègre le système MCTest et, dans sa version originale, dépend de modèles, de configurations et d'autres composants du *framework* Django. Comme ces dépendances ne sont pas disponibles dans l'environnement utilisé dans ce chapitre, le module doit être adapté pour être exécuté de manière indépendante.

Pour ce faire, le fichier est obtenu directement depuis le dépôt du projet à l'aide de la bibliothèque `requests`. Ensuite, des commandes `sed` sont employées pour supprimer les importations et les dépendances spécifiques à l'environnement Web, produisant ainsi une version autonome du module. Cette adaptation préserve l'implémentation des algorithmes de vision par ordinateur, permettant leur exécution et leur analyse sans avoir à installer ou configurer toute l'infrastructure du système 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)
```

Chaque commande `sed` supprime les importations spécifiques à l'environnement Django, rendant le fichier `CVMCTest.py` utilisable de manière indépendante dans ce chapitre.

```
# supprimer les lignes avec "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
```

Pourquoi supprimer les dépendances de Django ?

Dans l'implémentation originale, le fichier `CVMCTest.py` fait partie d'une application développée avec le *framework* Django et, de ce fait, importe des modèles, des configurations et d'autres composants spécifiques à cet environnement. Étant donné que ces éléments ne sont pas disponibles dans ce chapitre, le module ne peut pas être importé directement.
Les commandes `sed` suppriment uniquement ces dépendances, sans modifier les routines de Vision par Ordinateur implémentées dans le fichier. Ainsi, le module peut être exécuté de manière indépendante, en préservant le comportement des algorithmes présentés.
Cette procédure illustre un principe important de l'ingénierie logicielle : séparer la logique de l'application de l'infrastructure dans laquelle elle est intégrée, ce qui facilite la réutilisation, les tests et l'étude de composants spécifiques.

7.9.2 Extraction de la zone de réponses

Après la lecture de la feuille, la fonction `getAnswerArea` exécute automatiquement les étapes de détection des marqueurs de référence et de redressement par perspective présentées dans les sections précédentes. En conséquence, on obtient une image contenant uniquement la région destinée aux réponses, alignée et aux dimensions standardisées.

Cette standardisation simplifie les étapes ultérieures de traitement, car la localisation des champs de marquage devient connue et indépendante de la position originale de la feuille lors de la numérisation.

La Figure 7.13 présente la feuille de réponses originale en niveaux de gris, tandis que la Figure 7.14 montre la région de réponses obtenue après l'application de la fonction `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"[INGESTION] Page convertie : {myfile0}")
    pages.clear()
    img_color = mm.read(myfile0)
    img_inicial = mm.gray(img_color)
else:
    print("[AVERTISSEMENT] Fichier 'dados/provas_qrcode.pdf' introuvable.")
    print("        Utilisation de l'image publique skimage.data.page() comme substitut.")
    img_inicial = skdata.page()
```

```
mm.show(img_inicial)
```

```
[INGESTION] Page convertie : extra02.qrcode_p0.png
[INGESTION] Page convertie : extra02.qrcode_p1.png
[INGESTION] Page convertie : extra02.qrcode_p2.png
```

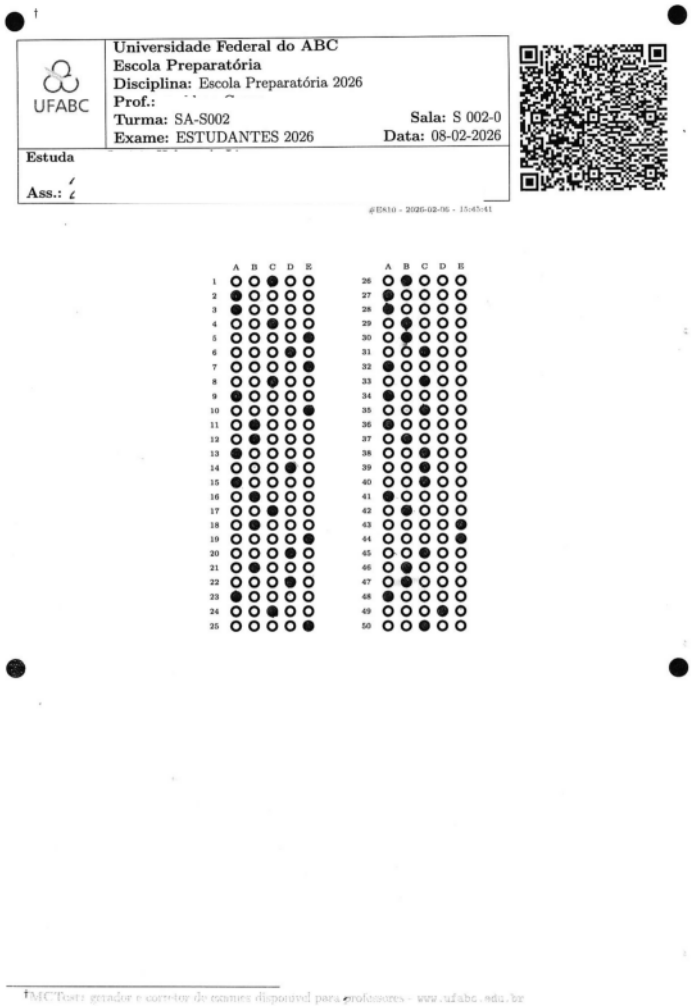


Figure 7.13: Image de la feuille de réponses en niveaux de gris chargée à partir du PDF rasterisé.

```
import CVMCTest
countPage = 0
img_getAnswerArea = CVMCTest.cvMCTest.getAnswerArea(img_inicial, countPage)
mm.show(img_getAnswerArea)
```

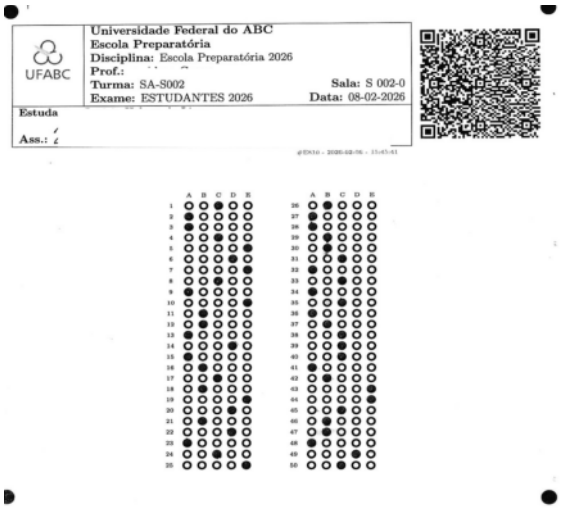


Figure 7.14: Zone de réponses extraite par *getAnswerArea* : région rectifiée contenant les cases de marquage.

Note de compatibilité : les versions récentes de NumPy (2.0) ont supprimé l'alias `np.int0`. Si `CVMCTest.py` utilise ce type, la commande ci-dessous applique la correction directement dans le fichier avant de le recharger :

```
!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.fr/cap06/CVMCTest.py'>
```

7.9.3 Segmentation du *QRCode*

Après l'extraction de la zone de réponses, MCTest effectue deux étapes préparatoires à la lecture des marquages : la segmentation du *QRCode* et la localisation des cadres contenant les questions.

La fonction `segmentQRcode` isole la région de l'image correspondant au *QRCode*, présentée dans la Figure 7.15. Ensuite, cette région est traitée par la fonction `getQRCode`, chargée de son décodage et de l'extraction des métadonnées de l'épreuve.

Si le *QRCode* ne peut pas être décodé, le traitement de la feuille se poursuit normalement. Les réponses de l'étudiant sont toujours lues et enregistrées dans le fichier CSV de sortie ; seules les informations obtenues à partir du *QRCode*, telles que l'identification de l'épreuve ou de l'étudiant, restent indisponibles.

```
import CVMCTest
imgQRcode = CVMCTest.cvMCTest.segmentQRcode(img_getAnswerArea, countPage)
mm.show(imgQRcode)
```

Figure 7.15: Région du *QRCode* isolée par *segmentQRcode* dans la zone de réponses rectifiée.

La fonction `CVMCTest.cvMCTest.getQRCode(img, countPage)` intègre les étapes de segmentation et de décodage du *QRCode*. Internement, elle utilise `CVMCTest.cvMCTest.decodeQRcode(imgQRcode)` pour interpréter la *chaîne* hexadécimale encodée dans le symbole et construire le dictionnaire `qr`, en plus de retourner l'indicateur logique `myFlagArea`, qui informe si la lecture a été réalisée avec succès.

Le dictionnaire `qr` regroupe les métadonnées de l'épreuve utilisées dans les étapes ultérieures du traitement. Ses principaux champs sont :

- **`date` :** identifiant temporel de l'épreuve, composé de la date de génération et d'un *timestamp* interne du MCTest.
- **`idClassroom`, `idExam` et `idStudent` :** identifiants de la classe, de l'épreuve et de l'étudiant.
- **`term` :** période scolaire.
- **`stylesheet` :** feuille de style utilisée lors de la génération du formulaire.
- **`var1` à `var5` :** nombre de questions à chaque niveau de difficulté.
- **`text` :** nombre de questions dissertatives.
- **`answer` :** nombre d'alternatives par question.
- **`numquest` :** nombre total de questions.
- **`correct`** et **`dbtext` :** champs remplis lors de la correction, contenant le corrigé et des informations supplémentaires.
- **`variations`** et **`variant` :** informations sur les versions de l'épreuve.

Ces métadonnées identifient l'épreuve et l'étudiant, permettant de sélectionner le corrigé correspondant et de paramétrer les étapes suivantes de lecture et de correction des réponses.

```
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'})
```

Ces métadonnées permettent d'identifier l'épreuve, de récupérer le corrigé correspondant et de paramétrer les étapes ultérieures du traitement.

Après le décodage du *QRCode*, le traitement revient à la zone des réponses pour localiser les cadres contenant les marquages de l'étudiant.

7.9.4 Localisation des cadres de réponses

L'image produite par `getAnswerArea` contient toute la région utile de la feuille, y compris l'en-tête, où se trouve le *QRCode*, ainsi que les cadres destinés aux réponses. Comme la lecture des marquages n'utilise que ces cadres, la région correspondant à l'en-tête est éliminée au moyen du recadrage `img_getAnswerArea[300:, :]`.

La Figure 7.16 présente cette région d'intérêt. Bien que ce recadrage soit par la suite utilisé par la fonction `findSquares` pour localiser les cadres de réponses, MCTest effectue initialement la lecture du *QRCode*, car celui-ci contient les métadonnées nécessaires pour identifier l'épreuve et configurer les étapes ultérieures du traitement.

```
img_getAnswerArea_aux = img_getAnswerArea[300:,:]
mm.show(img_getAnswerArea_aux)
```

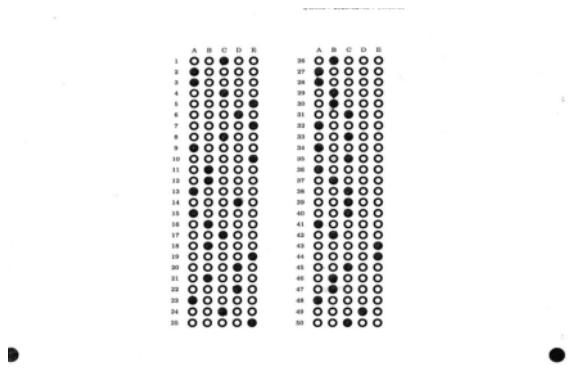


Figure 7.16: Recorte inférieur de la zone de réponses, concentrant les cadres de bulles à segmenter.

La fonction `findSquares` reçoit l'image de la zone de réponses et les métadonnées stockées dans `qr`, et retourne les coordonnées des cadres qui délimitent les groupes de questions.

Chaque élément de `rectSquares` contient les coordonnées des sommets supérieur gauche et inférieur droit d'un cadre de réponses, qui seront utilisées dans l'étape de segmentation des bulles.

```
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)]]]
```

7.9.5 Lecture automatique des réponses

Une fois les métadonnées de l'épreuve (`qr`) et les coordonnées des cadres de réponses (`rectSquares`) connues, le MCTest identifie automatiquement les alternatives cochées par l'étudiant.

Le code suivant intègre les étapes présentées précédemment. Pour chaque cadre délimité dans `rectSquares`, les fonctions `setColumns` et `setLines` estiment respectivement le nombre d'alternatives par question et le nombre de questions à partir de la distribution spatiale des bulles. Ensuite, `segmentAnswers` détermine l'alternative sélectionnée pour chaque question, et `setAnswersOneLine` regroupe les résultats de tous les cadres dans le champ `qr['answers']`.

Dans le mode de fonctionnement adopté dans ce chapitre, où le MCTest est exécuté indépendamment de sa base de données, le contenu de `qr['answers']` est comparé au corrigé stocké dans la première page du fichier PDF, correspondant au modèle d'épreuve sans énoncés utilisé dans les exemples.

La Figure 7.17 présente les cadres de réponses traités par l'algorithme, tandis que la sortie du programme affiche le contenu 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)
  # met les réponses de chaque tableau sur une ligne

mm.show(imgQ_all)
print(f"Réponses lues des {len(qr['answers'].split(","))} questions : \
      \n{qr['answers'][:-19]}...")
```

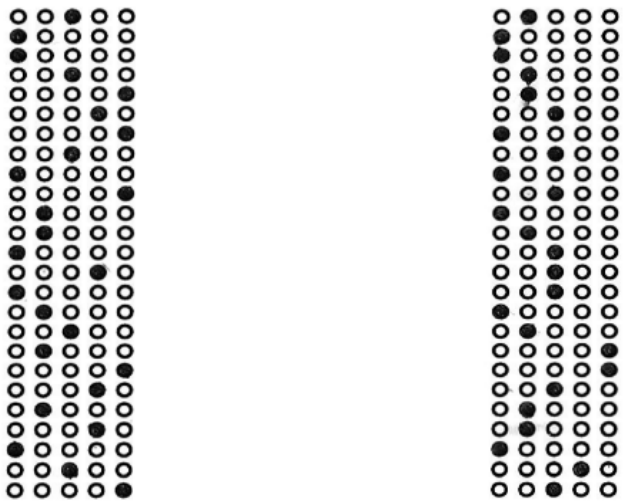


Figure 7.17: Réponses lues automatiquement par MCTest après segmentation et classification de toutes les bulles.

```
Réponses lues des 50 questions :
C,A,A,C,E,D,E,C,A,E,B,B,A,D,A,B,C,B,E,D,B,D,A,C,E,B,A,A,B,B,C,A,C,A,C,A,B,C,C,C,...
```

Relier les points

Le champ `qr['answers']` représente le résultat final du *pipeline* de vision par ordinateur présenté dans ce chapitre. Son obtention intègre toutes les étapes étudiées, depuis la rastérisation du document et la rectification géométrique jusqu'à l'extraction de la zone de réponses, le décodage du *QRCode*, la localisation des cadres et l'identification des alternatives marquées.
Ce *pipeline* illustre la transition d'un prototype vers un système en production. Dans MCTest, les algorithmes de vision par ordinateur restent essentiellement les mêmes ; les principales différences se concentrent sur des aspects d'ingénierie logicielle, tels que le traitement des exceptions, le support de différents modèles de formulaires, l'intégration avec la base de données, l'interface Web et les mécanismes d'audit et de maintenance.
Dans les expériences de ce chapitre, la première page du fichier PDF contient le corrigé de l'examen, tandis que les pages suivantes correspondent aux feuilles de réponses des étudiants. Après l'obtention de `qr['answers']`, MCTest compare automatiquement les réponses lues avec le corrigé pour calculer le score de chaque étudiant.
Lors de l'utilisation complète du système, les résultats de la correction sont consolidés dans un fichier CSV et envoyés à l'enseignant accompagnés d'un fichier compressé contenant des informations auxiliaires pour l'audit. Parmi ces fichiers figurent les extraits des questions où des marquages multiples ou d'autres situations nécessitant une révision manuelle ont été détectés. L'enseignant peut alors inspecter ces images, décider de l'interprétation la plus appropriée et, si nécessaire, mettre à jour le fichier CSV avant l'importation définitive des notes.

7.10 Inspection industrielle automatisée

L'inspection visuelle automatisée est une application de la vision par ordinateur dans laquelle des images de pièces ou de produits sont analysées pour vérifier le respect de critères de qualité préalablement définis. Dans une ligne de production, les images peuvent être obtenues par des caméras ou d'autres dispositifs d'acquisition et traitées automatiquement pour identifier des défauts, mesurer des dimensions ou vérifier la présence de composants.

La stratégie d'inspection dépend des caractéristiques du produit, du type de défaut d'intérêt et de la disponibilité d'une image de référence. Dans ce chapitre, deux approches classiques sont présentées :

- **Soustraction d'images :** compare l'image de la pièce inspectée avec une image de référence considérée comme exempte de défauts. Les régions où la différence d'intensité dépasse un seuil sont classées comme défauts possibles. Cette approche suppose que les images soient géométriquement alignées et aient été acquises dans des conditions d'éclairage similaires.
- **Analyse de texture :** utilise les caractéristiques de la texture de la surface pour identifier les régions dont l'apparence diffère du motif attendu, sans nécessiter d'image de référence. Cette approche est adaptée aux matériaux présentant une texture approximativement homogène, comme les tissus, les papiers et les surfaces métalliques.

Dans les sections suivantes, ces deux stratégies sont illustrées au moyen d'exemples construits à partir d'images de la bibliothèque `skimage.data`. L'objectif est de présenter les principes de fonctionnement de chaque approche dans des expériences pouvant être reproduites intégralement par le lecteur.

7.10.1 Références et *Datasets* publics

Dans les applications d'inspection industrielle, la performance des algorithmes de détection de défauts est souvent évaluée sur des *datasets* publics, qui fournissent des images représentatives et, dans de nombreux cas, des annotations de référence (*ground truth*). Dans ce chapitre, cependant, les exemples utilisent des images synthétiques dérivées de `skimage.data` (voir Figure 7.18), permettant de reproduire toutes les expériences sans dépendance à des bases de données externes.

Pour des études plus complètes et la comparaison entre algorithmes, on distingue les *datasets* publics suivants :

- **MVTec *Anomaly Detection Dataset* (MVTec AD) :** ensemble d'images d'objets et de textures, contenant des échantillons sans défauts et avec défauts, accompagnés de masques de segmentation au niveau du pixel pour les images anormales (BERGMANN, 2019). Disponible sur : https://www.mvtec.com/company/research/datasets/mvtec-ad.
- **Kolektor *Surface-Defect Dataset* (KolektorSDD) :** ensemble d'images de composants industriels avec des défauts de surface annotés, utilisé dans les études de détection et de segmentation de défauts (TABERNIK, 2020). Disponible sur : https://www.vicos.si/resources/kolektorsdd/.
- **NEU *Surface Defect Database* :** ensemble d'images de surfaces d'acier laminé, organisé en six catégories de défauts de surface, fréquemment utilisé dans l'évaluation des méthodes de classification et de détection (SONG, 2013). Disponible sur : 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

# Image de référence (produit sans défaut)
product_color = data.coffee()
product_gray = color.rgb2gray(product_color)

# Insertion de défaut simulé : rayure sombre de 10×100 px
defect_image = np.copy(product_gray)
defect_image[100:110, 200:300] = 0.1

# Détection par soustraction et seuillage
difference = np.abs(product_gray - defect_image)
defect_threshold = 0.15          # ajustable selon l'application
detected_defect = (difference > defect_threshold).astype(np.uint8) * 255

mm.show(
    [product_gray, defect_image, detected_defect],
    titles=["Référence", "Avec défaut", "Défaut détecté"],
```

```
    cols=3,
    figsize=(12, 4)
)

status = "Defeito detectado." if detected_defect.any() else "Produto conforme."
print(status)
```

Figure 7.18: Détection de défaut par soustraction d'image : produit de référence, image avec défaut simulé et masque d'anomalie détectée.

```
Defeito detectado.
```

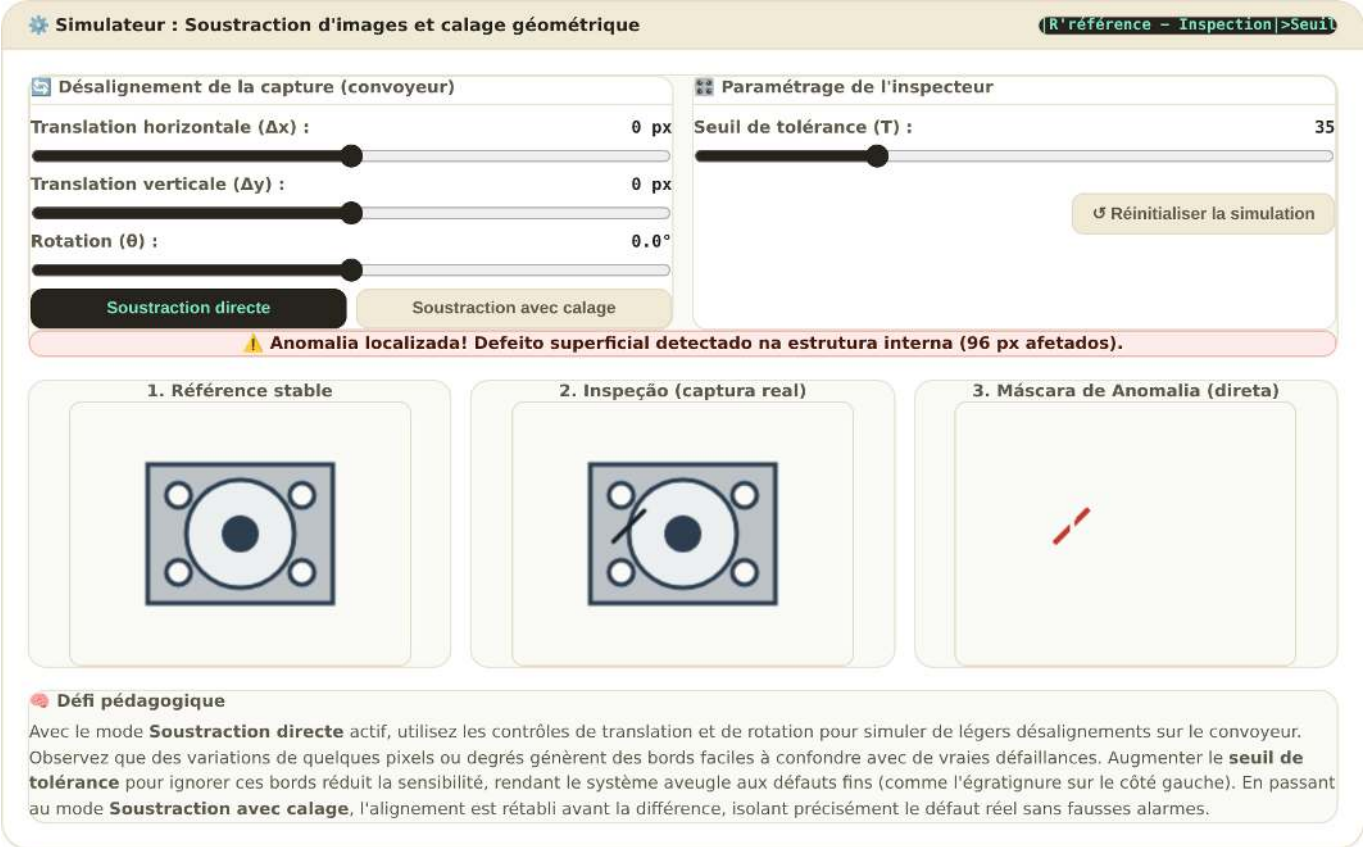
i Soustraction d'images : enregistrement géométrique et principe de fonctionnement

La soustraction d'images suppose que l'image d'inspection soit géométriquement alignée sur l'image de référence. Les différences de positionnement, de rotation, d'échelle ou de perspective produisent des régions de différence qui peuvent être confondues avec des défauts.
Dans les applications à acquisition contrôlée, cet alignement est obtenu lors de la capture au moyen de gabarits mécaniques, de tapis roulants et de caméras fixes, ce qui permet de comparer directement des images successives. Un exemple est l'inspection d'une boîte à outils toujours positionnée dans la même orientation afin de vérifier l'absence d'un quelconque élément.
Lorsque ce contrôle n'est pas possible, on emploie l'**enregistrement d'images**, qui estime une transformation géométrique pour compenser les différences de translation, de rotation, d'échelle et, si nécessaire, de perspective.
Après l'enregistrement, on effectue la comparaison pixel par pixel entre les deux images. Dans les régions sans modification, les différences d'intensité tendent à être proches de zéro ; là où un défaut existe, des différences locales apparaissent et peuvent être mises en évidence par seuillage. L'utilisation de la différence absolue permet de détecter aussi bien les défauts plus clairs que plus sombres que la référence.
La performance de la méthode dépend principalement de la qualité de l'alignement géométrique et du choix du seuil utilisé pour séparer les petites variations d'acquisition des différences associées aux défauts.
La Figure 7.19 illustre l'effet du désalignement entre les images et l'importance de l'enregistrement géométrique avant l'application de la soustraction.

7.10.2 Détection de défauts par analyse de texture

Dans les applications où il n'existe pas d'image de référence, la détection de défauts peut être basée sur les caractéristiques de texture de la surface. Dans ce cas, on cherche à identifier les régions dont l'apparence diffère du motif prédominant du matériau.

Dans cet exemple, on utilise la **variance locale** comme mesure d'hétérogénéité. Pour chaque position de l'image, on calcule la variance des niveaux d'intensité dans un voisinage de dimensions fixes. Les régions à faible variance tendent à présenter une texture plus uniforme, tandis que des altérations locales, telles que des rayures, des taches ou des imperfections, peuvent produire des valeurs plus élevées de cette mesure.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap06/sim-06-industrial.html

Figure 7.19: Simulateur interactif d'inspection industrielle par soustraction d'images : contrôlez les distorsions géométriques de désalignement (enregistrement) et le seuil de détection pour observer l'impact sur les faux positifs.

La Figure 7.20 illustre cette procédure en utilisant l'image `skimage.data.brick()`. Dans un premier temps, on calcule la carte de variance locale au moyen d'une fenêtre glissante. Ensuite, on applique un seuillage pour mettre en évidence les régions dont la variance dépasse la valeur spécifiée, identifiant ainsi les zones d'intérêt potentielles pour l'inspection.

Modélisation mathématique

Considérons une image en niveaux de gris représentée par

$$f : \Omega \subset \mathbb{Z}^2 \to \mathbb{R},$$

où Ω est le domaine de l'image et $f(x, y)$ représente l'intensité du pixel aux coordonnées (x, y). Dans les images 8 bits, ces intensités appartiennent à l'intervalle $[0, 255]$. Dans cet exemple, elles ont toutefois été normalisées dans l'intervalle $[0, 1]$, sans modifier le fonctionnement de l'algorithme.

Pour chaque position de l'image, on considère un voisinage carré $W_{x,y}$ de dimension 15×15 pixels.

La moyenne locale est donnée par

$$\mu(x, y) = \frac{1}{|W_{x,y}|} \sum_{(u,v)\in W_{x,y}} f(u, v),$$

et la variance locale est calculée par

$$\sigma^2(x, y) = \frac{1}{|W_{x,y}|} \sum_{(u,v)\in W_{x,y}} f(u, v)^2 - \mu(x, y)^2.$$

Dans l'implémentation suivante, ces deux moyennes sont obtenues par la fonction `cv2.blur`,

```
mean  = cv2.blur(img, (15,15))
mean2 = cv2.blur(img**2, (15,15))
var   = mean2 - mean**2
```

Ensuite, on calcule la différence entre les cartes de variance de l'image de référence et de l'image inspectée,

$$D(x, y) = \left|\sigma_d^2(x, y) - \sigma_r^2(x, y)\right|,$$

où $\sigma_r^2(x, y)$ et $\sigma_d^2(x, y)$ sont respectivement les variances locales de l'image de référence et de l'image contenant le défaut. Après la normalisation de la carte $D(x, y)$, on applique un seuillage pour obtenir le masque des anomalies possibles.

```
import numpy as np
import cv2
from skimage import data as skdata
from morph import mm

# Image de texture uniforme (brique)
texture = skdata.brick().astype(np.float32) / 255.0

# Insertion de défaut synthétique : tache claire 20×80 px
texture_defect = np.copy(texture)
texture_defect[60:80, 80:160] = 0.95

# Carte de variance locale (fenêtre 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)

# Normalise et seuille
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=["Texture originale", "Avec défaut", "Δ variance locale", "Anomalie détectée"],
    cols=4,
    figsize=(16, 4)
)

status = "Defeito de textura detectado." if mask.any() else "Superfície conforme."
print(status)
```

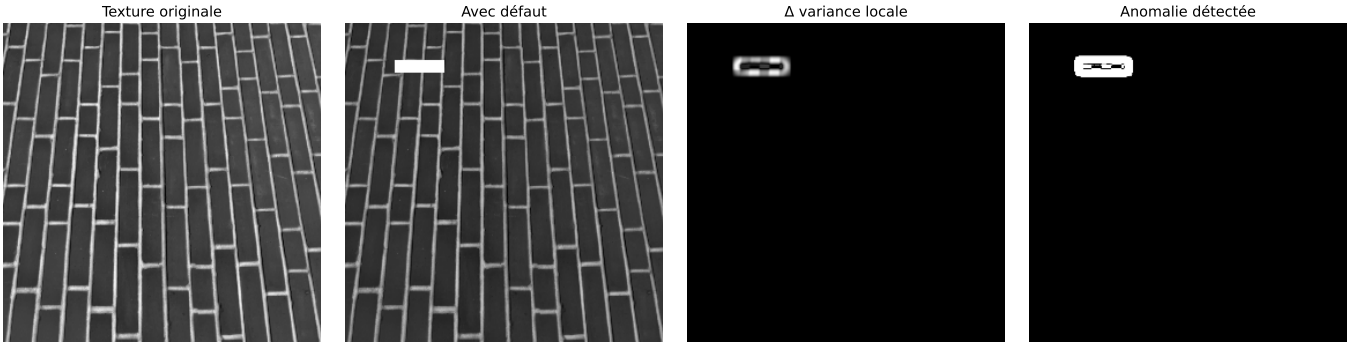


Figure 7.20: Détection d'hétérogénéité de texture : carte de variance locale et masque d'anomalie.

```
Defeito de textura detectado.
```

Pourquoi cela fonctionne-t-il ? — Analyse de texture

La variance locale mesure la dispersion des intensités dans un voisinage de l'image. Dans les régions où la texture reste uniforme, cette mesure tend à varier peu. Lorsqu'un défaut modifie le motif de la surface, la distribution des intensités se modifie également, produisant des différences dans la variance locale.
Dans ce chapitre, la détection est réalisée en comparant les cartes de variance de l'image de référence et de l'image avec défaut. Après normalisation, on applique un seuillage pour mettre en évidence les régions où cette différence dépasse une valeur spécifiée.
Les principaux paramètres de la méthode sont la taille de la fenêtre utilisée pour le calcul de la variance et le seuil employé pour la segmentation. Des fenêtres plus petites sont plus sensibles aux détails fins, tandis que des fenêtres plus grandes produisent des cartes plus lisses et peuvent réduire la réponse aux défauts de petites dimensions.

7.11 Résumé

Dans ce chapitre, des méthodes de Vision par Ordinateur appliquées à l'analyse de documents et à l'inspection visuelle automatisée ont été présentées. Les principales techniques étudiées étaient :

- **Prétraitement de documents :** application de la normalisation du fond, de l'égalisation adaptative (CLAHE) et du seuillage par Otsu pour réduire les effets d'un éclairage non uniforme et améliorer la segmentation du texte.
- **Reconnaissance optique de caractères (OCR) :** conversion d'images de documents en texte codé au moyen du Tesseract OCR, mettant en évidence l'influence du prétraitement sur la qualité de la reconnaissance.
- **Traduction automatique :** application de techniques de traitement du langage naturel pour traduire le texte obtenu par l'OCR.
- **Redressement géométrique de documents :** utilisation du détecteur de contours de Canny, de la Transformée de Hough et de transformations projectives pour corriger la perspective de documents numérisés.
- **Localisation et redressement de formulaires :** emploi d'opérations morphologiques, d'analyse de contours et de transformation de perspective pour identifier des marqueurs de référence et extraire automatiquement des régions d'intérêt.
- **Lecture de codes bidimensionnels et unidimensionnels :** détection et décodage de *QRCodes* et de codes-barres pour l'identification automatique de documents et de métadonnées.
- **Reconnaissance optique de marques (OMR) :** lecture automatisée de formulaires et de feuilles de réponses, illustrée par une étude de cas du système MCTest.
- **Inspection industrielle :** détection de défauts par comparaison avec une image de référence et par analyse de la variance locale.

Tout au long du chapitre, les algorithmes ont été implémentés et évalués avec des images de la bibliothèque `skimage.data` et avec des documents réels, permettant de reproduire les expériences présentées.

Prochaines étapes

Les méthodes présentées dans ce chapitre montrent comment des techniques de Traitement Numérique des Images, de Vision par Ordinateur et de Traitement du Langage Naturel peuvent être intégrées dans des *pipelines* pour l'analyse automatisée de documents.

Dans le prochain chapitre, seront étudiées des techniques d'**extraction de caractéristiques** et de **reconnaissance de formes**, avec un accent sur les descripteurs capables de représenter des images par des attributs numériques pour la comparaison, la classification et la reconnaissance automatique. Ces concepts constituent la base des chapitres consacrés à l'apprentissage automatique et à l'apprentissage profond appliqués à la Vision par Ordinateur.

7.12 🤖 Utilisation de Gemini Notebook comme tuteur complémentaire

Pour accompagner l'étude de ce chapitre, il est recommandé d'utiliser **Gemini Notebook** comme tuteur complémentaire. L'outil recourt à des modèles d'intelligence artificielle pour répondre aux questions, élaborer des résumés et expliquer des concepts à partir des documents fournis comme source de consultation, permettant à l'étudiant de réviser le contenu de manière interactive.

🎓 Étudiez avec le tuteur intelligent

🚀 ACCÉDER À Gemini Notebook : CHAPITRE 06

🌐 **Langue et langage de programmation**

Le projet de ce chapitre dans Gemini Notebook a été construit uniquement avec le texte en **portugais** et les exemples de code en **Python**. Si vous étudiez à partir de l'édition en anglais ou en français, ou

que vous suivez le parcours en C++, les réponses du tuteur peuvent ne pas correspondre exactement à la version que vous lisez.

⚠️ **Usage critique des réponses**

Les réponses générées par Gemini Notebook sont produites automatiquement par un modèle d'intelligence artificielle et peuvent comporter des omissions ou des imprécisions. Pour cette raison, elles doivent être utilisées comme matériel de soutien, et non comme substitut à l'étude du chapitre.
En cas de doute, consultez le texte de cet ouvrage, exécutez les exemples présentés et, si nécessaire, complétez la consultation avec des livres, des articles scientifiques et d'autres sources académiques fiables.

7.13 Liste d'exercices

Les exercices suivants consolident les concepts présentés dans ce chapitre à travers des adaptations, des expérimentations et des extensions des algorithmes développés tout au long du texte.

1. **(10%)** Étudiez l'influence de l'angle d'inclinaison dans l'étape de *deskew*. Générez des versions pivotées de l'image `skimage.data.page()` pour des angles entre -10° et 10°, appliquez l'algorithme présenté dans le chapitre et comparez l'angle estimé avec l'angle utilisé lors de la rotation. Présentez les résultats dans un tableau et discutez de la précision de la méthode.

2. **(15%)** Appliquez une normalisation du fond et un CLAHE (en utilisant au moins trois combinaisons de `clipLimit` et `tileGridSize`) sur l'image `skimage.data.page()` dégradée artificiellement avec un gradient d'éclairage et une ombre latérale. Segmentez chaque version à l'aide de la méthode d'Otsu et comparez les résultats en utilisant le nombre de composantes connexes parasites et la métrique IoU par rapport à un masque de référence construit manuellement.

3. **(15%)** Étudiez la sensibilité du filtrage par circularité, $C = \frac{4\pi A}{P^2}$, dans la détection des marqueurs circulaires. À l'aide du simulateur de la Figure 7.9, générez des disques synthétiques avec un bruit géométrique croissant et évaluez les seuils $C \in \{0{,}5,\ 0{,}6,\ 0{,}7,\ 0{,}8\}$. Présentez un tableau reliant le seuil au nombre de faux positifs et de faux négatifs et discutez du compromis entre sensibilité et spécificité.

4. **(15%)** À partir des quatre marqueurs détectés, implémentez la rectification par perspective à l'aide de `cv2.getPerspectiveTransform` et `cv2.warpPerspective`. Ensuite, perturbez artificiellement les coordonnées des points de contrôle avec un bruit gaussien d'écart-type $\sigma \in \{1, 3, 5\}$ pixels et évaluez l'erreur de reprojection obtenue après l'homographie inverse.

5. **(15%)** Adaptez le *pipeline* d'acquisition et de rectification développé dans ce chapitre pour traiter des documents contenant des codes-barres linéaires en remplacement des *QRCodes*. Rasterisez le PDF avec `pdf2image` à 300 DPI, appliquez le *deskew*, décodez le symbole avec `pyzbar` et présentez l'image rectifiée accompagnée de la séquence de caractères obtenue.

6. **(15%)** Étendez la lecture des bulles du MCTest pour identifier trois situations : **OK**, **BLANC** (aucune alternative cochée) et **DOUBLE MARQUAGE** (deux alternatives ou plus au-dessus d'un seuil de remplissage). Évaluez au moins trois valeurs de ce seuil, présentez les résultats dans un `pandas.DataFrame` et discutez de son influence sur la classification des réponses.

7. **(15%)** Construisez un *pipeline* d'inspection industrielle combinant la soustraction d'images et l'analyse de variance locale de la texture sur un ensemble d'images synthétiques contenant des défauts simulés. Pour chaque image, générez un masque de référence (*ground truth*), calculez la métrique IoU (*Intersection over Union*) des deux approches pour différents seuils de décision et présentez les résultats dans des tableaux et des visualisations produites avec `mm.show`.

8. **(Bonus – 10%)** Implémentez manuellement l'estimation de l'angle d'inclinaison sans utiliser `cv2.HoughLines` ou `cv2.HoughLinesP`. À partir de la carte des contours obtenue par le détecteur de Canny, construisez l'accumulateur de la Transformée de Hough, $\rho = x\cos\theta + y\sin\theta$, pour

$\theta \in [-45°, 45°]$, identifiez les maxima de l'accumulateur et estimez l'inclinaison par la médiane des droites détectées. Comparez les résultats avec l'implémentation d'OpenCV et discutez de l'influence des droites parasites sur l'estimation finale.

Références du chapitre

Le fondement théorique et les études de cas présentés dans ce chapitre s'appuient sur les références suivantes :

- Gonzalez (2018), pour les fondements de la détection de contours, du seuillage, de la segmentation, des opérations morphologiques, de la reconnaissance optique de caractères et des transformations géométriques appliquées à l'analyse de documents.
- Szeliski (2022), pour la transformée de Hough, le recalage et l'alignement d'images, les transformations projectives (homographies) et les principes de l'inspection visuelle automatisée.
- Bradski (2008), pour l'utilisation de la bibliothèque OpenCV dans les étapes de détection de contours, de transformée de Hough, de transformations géométriques, d'analyse de contours et de décodage de *QR codes*.
- Smith (2007) et Smith (2013), pour l'architecture, le fonctionnement et l'évolution du moteur de reconnaissance optique de caractères **Tesseract OCR**, utilisé dans les exemples d'OCR présentés dans ce chapitre.
- Bahdanau (2015) et Vaswani (2017), pour les principes de la traduction automatique fondée sur les réseaux de neurones, y compris les mécanismes d'attention et les architectures *transformer*.
- Zampirolli (2023), pour la description du système MCTest, utilisé comme étude de cas d'un *pipeline* complet pour la lecture et la correction automatisées de feuilles de réponses.
- Bergmann (2019), Tabernik (2020) et Song (2013), pour les *jeux de données* publics d'inspection industrielle **MVTec AD**, **KolektorSDD** et **NEU Surface Defect Database**, utilisés comme référence pour l'évaluation et la comparaison des algorithmes de détection de défauts.

Exécuter Colab Ouvrir GitHub

7.14 💻 Partie pratique avec exercices de programmation

Les exercices de programmation (EP) de cette section complètent les concepts présentés tout au long du chapitre 6 à travers la mise en œuvre d'algorithmes liés à l'inspection industrielle et à l'analyse de documents. L'objectif est de consolider les fondamentaux étudiés, en reproduisant, à échelle réduite, les étapes d'un *pipeline* typique de Vision par Ordinateur.

Contrairement aux chapitres précédents, dont les exercices mettaient l'accent sur des opérations plus directement liées aux données d'image, les EP de ce chapitre se concentrent sur les **grandeurs intermédiaires** produites au cours du traitement, telles que les aires, les périmètres, la circularité, les angles de droites, les degrés de remplissage de bulles, les cartes de variance et les cartes de différence. Cette approche permet de comprendre et de valider chaque étape du *pipeline* de manière indépendante, sans dépendre de bibliothèques spécialisées pour l'acquisition d'images, la détection de marqueurs ou le décodage de codes — à l'exception de l'exercice de clôture du chapitre (EP06_08), qui introduit délibérément l'utilisation d'OpenCV pour la segmentation et le décodage réel d'un *QRCode*, bouclant ainsi la boucle entre les concepts théoriques et les outils employés en pratique.

Les exercices suivent la même séquence conceptuelle que le chapitre, par ordre croissant de complexité. Dans un premier temps, sont abordées les métriques d'évaluation de segmentation, utilisées pour quantifier la qualité des masques binaires. Ensuite, sont étudiés les critères géométriques pour la sélection de marqueurs, la classification de marquages dans des formulaires et l'estimation de l'inclinaison de documents au moyen de

la Transformée de Hough. Dans la partie finale, les exercices explorent la normalisation de l'éclairage, la détection de défauts par analyse de texture et l'intégration entre le recalage géométrique et la soustraction d'images dans un *pipeline* simplifié d'inspection industrielle.

Chaque exercice représente une étape isolée d'un système réel de Vision par Ordinateur, permettant de valider individuellement des concepts qui, dans les applications industrielles, sont combinés en un seul *pipeline* d'inspection.

Légende de difficulté

Niveau	Signification	EP
🟢	Très facile / facile — mise en œuvre d'un seul concept ou d'un algorithme simple	EP06_01, EP06_02
🟡	Facile–moyen — traitement de multiples cas ou utilisation de critères statistiques simples	EP06_03, EP06_04
🟠	Moyen — traitement matriciel point à point	EP06_05
🔴	Difficile — traitement matriciel avec opérations de voisinage (fenêtre glissante)	EP06_06
🟣	Très difficile — intégration de multiples étapes d'un *pipeline* de Vision par Ordinateur	EP06_07
⚫	Spécial — utilisation d'une bibliothèque spécialisée (`cv2`) pour la segmentation géométrique et le décodage réel de code-barres/QRCode	EP06_08

❗ Directives pour la résolution des exercices de programmation

Sauf indication contraire, tous les exercices utilisent la convention de coordonnées matricielles `[ligne][colonne]`, avec l'origine en $(0, 0)$ dans le coin supérieur gauche de l'image.
Lorsqu'un arrondi numérique est nécessaire, on doit utiliser l'arrondi standard à l'entier le plus proche (*round half away from zero*, avec `np.floor(img + 0.5)`). Les comparaisons avec des seuils (par exemple, circularité, variance, différence d'intensité ou degré de remplissage) doivent être considérées comme **strictes** (`>`), sauf si l'énoncé spécifie explicitement un autre critère.
Chaque exercice a été conçu pour mettre l'accent sur un concept spécifique présenté dans le chapitre. Il est recommandé de mettre en œuvre initialement la solution de manière directe et, seulement après sa validation, de rechercher des alternatives plus efficaces ou plus générales.

🎯 Objectif de ce cahier

Ce cahier a été élaboré pour accompagner le développement, la validation et les tests des solutions des **Exercices de Programmation (EPs)** dans un environnement interactif, tel que Google Colab ou Jupyter Notebook. Après avoir vérifié le fonctionnement de l'implémentation avec les cas de test présentés, le code peut être soumis à Moodle pour l'évaluation officielle.

Download

Exécutez la cellule suivante pour obtenir les fichiers `morph.py` et `testsuite.py`, utilisés par les exercices de ce chapitre.

```
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
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0 | TestSuite : 1.1.2
```

Exécution des Tests

Après avoir implémenté la solution, exécutez `TestSuite("EP06_01.extensão").run()` dans une nouvelle cellule, en remplaçant `extensão` par le langage utilisé (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). Le système récupère automatiquement les cas de test du dépôt du cours, exécute le programme et affiche le résultat de l'évaluation.

En Python, il est également possible de tester la solution directement à partir d'une *chaîne de caractères*, sans avoir besoin d'enregistrer le code dans un fichier. Pour cela, stockez le programme dans une variable et utilisez la méthode `run_code` :

```
codigo = """
# ... votre code ici ...
"""

TestSuite("EP06_01").run_code(codigo)
```

7.14.1 EP06_01 🟢 Évaluation de Segmentation par IoU (*Intersection over Union*)

Tout au long de ce chapitre, plusieurs étapes du *pipeline* produisent des **masques binaires**, comme dans la segmentation de documents, la localisation de *QRCodes* et la détection de défauts. Pour évaluer objectivement la qualité de ces segmentations, il est nécessaire de les comparer à un masque de référence (*ground truth*).

L'une des métriques les plus utilisées à cette fin est l'**IoU** (*Intersection over Union*, ou Intersection sur Union), définie comme le rapport entre l'aire de l'intersection et l'aire de l'union de deux masques binaires. Plus la valeur de l'IoU est élevée, plus la concordance entre la segmentation produite par l'algorithme et la référence est grande.

7.14.1.1 📋 Directives d'Implémentation

1. **Dimensions :** Lire les entiers L (nombre de lignes) et C (nombre de colonnes).
2. **Masque de référence :** Lire les $L \times C$ éléments binaires (0 ou 1) de la matrice `ref`.
3. **Masque prédit :** Lire les $L \times C$ éléments binaires (0 ou 1) de la matrice `pred`.
4. **Intersection :** Compter le nombre de positions (i, j) pour lesquelles `ref[i][j] = 1` et `pred[i][j] = 1`.
5. **Union :** Compter le nombre de positions (i, j) pour lesquelles `ref[i][j] = 1` ou `pred[i][j] = 1`.
6. **Cas dégénéré :** Si l'union est égale à 0, définir IoU = 1,0, car les deux masques sont vides.
7. **Calcul :** Si l'union est supérieure à zéro, calculer

$$\text{IoU} = \frac{|\text{Intersecao}|}{|\text{Uniao}|}.$$

8. **Classification :** Déterminer la classification qualitative en utilisant la valeur de l'IoU **avant** l'arrondi.
9. **Arrondi :** Afficher l'IoU avec quatre décimales.
10. **Sortie :** Imprimer, dans cet ordre, l'intersection, l'union, l'IoU et la classification.

7.14.1.2 📌 Contraintes Computationnelles

- Si l'union est égale à 0, la division ne doit pas être effectuée ; l'IoU doit être définie comme 1,0.
- Les plages de classification utilisent des comparaisons non strictes ($\geq$).
- La classification doit être effectuée en utilisant la valeur de l'IoU en précision complète, avant l'arrondi pour l'affichage.

7.14.1.3 🧠 Fondements Théoriques

L'IoU est définie par

$$\text{IoU} = \frac{|R \cap P|}{|R \cup P|},$$

où :

- R représente l'ensemble des pixels appartenant au masque de référence ;
- P représente l'ensemble des pixels appartenant au masque prédit ;
- $|R \cap P|$ correspond au nombre de pixels appartenant simultanément aux deux masques ;
- $|R \cup P|$ correspond au nombre de pixels appartenant à au moins l'un des masques.

Plage d'IoU	Classification	Interprétation
IoU $\geq$ 0,90	`EXCELENTE`	Concordance très élevée entre les masques.
0,70 $\leq$ IoU $<$ 0,90	`BOM`	Petites différences entre les masques.
0,50 $\leq$ IoU $<$ 0,70	`ACEITAVEL`	Concordance partielle entre les masques.
IoU $<$ 0,50	`RUIM`	Faible concordance entre les masques.

L'IoU dépend uniquement du chevauchement entre les masques et est donc indépendante de la taille de l'image.

7.14.1.4 📦 Spécification d'Entrée et de Sortie (VPL)

Entrée :

- Ligne 1 : entier L.
- Ligne 2 : entier C.
- L lignes suivantes : éléments binaires (0 ou 1) de la matrice `ref`.
- L lignes suivantes : éléments binaires (0 ou 1) de la matrice `pred`.

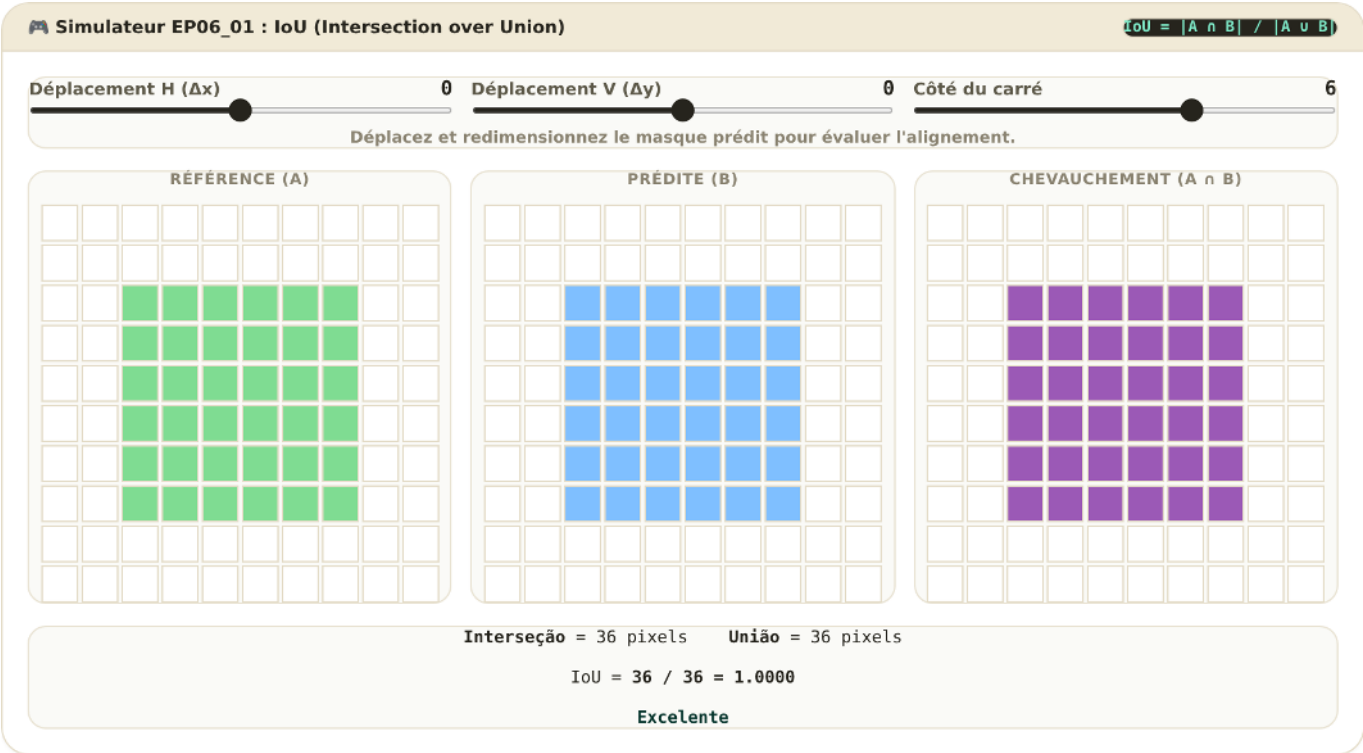
Sortie :

- Ligne 1 : `Intersecao: X`
- Ligne 2 : `Uniao: Y`
- Ligne 3 : `IoU: Z`
- Ligne 4 : `Classificacao: NOME`

La valeur de `IoU` doit être imprimée avec quatre décimales.

7.14.1.5 📌 Exemples

Entrée	Sortie	Observation
2 2 1 1 0 0 1 0 0 0	Intersecao: 1 Uniao: 2 IoU: 0.5000 Classificacao: ACEITAVEL	La moitié de la région de référence a été correctement segmentée.
2 2 0 0 0 0 0 0 0 0	Intersecao: 0 Uniao: 0 IoU: 1.0000 Classificacao: EXCELENTE	Les deux masques sont vides ; par convention, IoU = 1,0.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap06/sim-ep0601.html

Figure 7.21: Simulateur EP06_01 : IoU entre masque de référence et masque prédit

```
%%writefile EP06_01.py
# Code Python
```

```
Overwriting EP06_01.py
```

```
TestSuite("EP06_01.py").run()
```

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

7.14.2 EP06_02 🟢 Filtrage des marqueurs par circularité

Après la segmentation d'une image, il est courant que plusieurs composants connexes soient identifiés. Dans des applications telles que la rectification de documents, seuls certains de ces composants correspondent aux marqueurs de référence utilisés pour l'alignement de l'image. Un critère fréquemment employé pour sélectionner ces marqueurs est la **circularité**, qui mesure à quel point la forme d'un composant est proche d'un cercle.

Dans cet exercice, chaque composant est décrit par son aire A et son périmètre P. L'objectif est de calculer sa circularité et de décider, à partir d'un seuil fourni, si le composant doit être accepté ou rejeté comme candidat marqueur.

7.14.2.1 📋 Directives d'implémentation

1. **Quantité :** Lire l'entier N (nombre de candidats) et le seuil de circularité C_{limiar} (nombre réel).
2. **Données des candidats :** Pour chacun des N candidats, lire l'aire A (entier) et le périmètre P (nombre réel).
3. **Circularité :** Calculer $C = \frac{4\pi A}{P^2}$, où :

- A est l'aire du composant ;
- P est le périmètre du composant ;
- C est la circularité.

4. **Cas dégénéré :** Si $P = 0$, considérer $C = 0$ et classer directement le candidat comme `REJETÉ`.
5. **Classification :** Si $C > C_{\text{limiar}}$, classer le candidat comme `ACCEPTÉ` ; sinon, le classer comme `REJETÉ`.
6. **Arrondi :** Afficher la valeur de C avec quatre décimales.
7. **Sortie :** Pour chaque candidat, imprimer la valeur de C suivie de la classification. À la fin, imprimer le nombre total de candidats acceptés.

7.14.2.2 📌 Contraintes computationnelles

- Utiliser la constante π de la bibliothèque standard du langage (par exemple, `math.pi`), sans approximations.
- La comparaison doit être effectuée avec la valeur de C en précision complète, avant l'arrondi pour l'affichage.
- Le critère d'acceptation est strict ($C > C_{\text{limiar}}$).
- Si $P = 0$, la division ne doit pas être effectuée.

7.14.2.3 🧠 Fondement théorique

La circularité est un descripteur géométrique défini par $C = \frac{4\pi A}{P^2}$, où :

- A est l'aire du composant ;
- P est le périmètre du composant ;
- C est la circularité.

Pour un cercle parfait, $C = 1$. À mesure que la forme devient plus allongée ou irrégulière, le périmètre croît plus rapidement que l'aire, réduisant la valeur de C.

Forme	Circularité approximative	Interprétation
Cercle	1,0000	Forme circulaire.
Carré	0,7854	Forme approximativement compacte.
Forme allongée ou irrégulière	$C \ll 1$	Faible circularité.
$P = 0$	0 (convention adoptée)	Contour dégénéré.

La circularité est invariante par translation, rotation et mise à l'échelle, et est largement utilisée pour distinguer les composants approximativement circulaires des autres formats.

7.14.2.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : entier N.
- Ligne 2 : nombre réel C_{limiar}.
- Les N lignes suivantes : aire A (entier) et périmètre P (réel), séparés par un espace.

Sortie :

- Une ligne pour chaque candidat, au format `C ACCEPTÉ` ou `C REJETÉ`, avec C présenté avec quatre décimales.
- Dernière ligne : `Total acceptés : X`.

7.14.2.5 📌 Exemples

Entrée	Sortie	Observation
3 0.6 78 31.4 100 40 50 60	0.9941 ACCEPTÉ 0.7854 ACCEPTÉ 0.1745 REJETÉ Total acceptés : 2	Candidat approximativement circulaire, forme compacte et forme allongée.
1 0.9 10 0	0.0000 REJETÉ Total acceptés : 0	Périmètre nul : contour dégénéré.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap06/sim-ep0602.html

Figure 7.22: Simulateur EP06_02 : Filtre de Marqueurs par Circularité

```
%%writefile EP06_02.py
# Code Python
```

```
Overwriting EP06_02.py
```

```
TestSuite("EP06_02.py").run()
```

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

7.14.3 EP06_03 🟡 Classification des Marques sur les Feuilles de Réponses (OMR)

Après la correction de la feuille et la segmentation des cadres de réponses, le MCTest estime, pour chaque bulle, un **degré de remplissage**, représenté par une valeur entre 0 et 100. À partir de ces valeurs, le système

doit déterminer automatiquement l'alternative marquée, en identifiant également les questions blanches et les cas de marques multiples.

Dans cet exercice, vous implémenterez cette étape de décision du *pipeline* OMR. La classification dépend d'un seuil de remplissage : de petites variations de cette valeur peuvent modifier le résultat de la lecture automatique.

7.14.3.1 📋 Directives d'Implémentation

1. **Paramètres :** Lire les entiers Q (nombre de questions) et K (nombre d'alternatives par question, avec $2 \leq K \leq 26$) ainsi que le seuil de remplissage Th (nombre réel entre 0 et 100).
2. **Degrés de remplissage :** Pour chacune des Q questions, lire les K valeurs réelles correspondant aux alternatives `A`, `B`, `C`, …, dans l'ordre de saisie.
3. **Comptage des marques :** Pour chaque question, compter combien d'alternatives ont un degré de remplissage **strictement supérieur** à Th.
4. **Classification :**
   - Si aucune alternative ne dépasse Th, classer la question comme `BRANCO`.
   - Si exactement une alternative dépasse Th, imprimer la lettre correspondante (`A`, `B`, `C`, …).
   - Si deux alternatives ou plus dépassent Th, classer la question comme `DUPLA_MARCACAO`.
5. **Sortie par question :** Imprimer, dans l'ordre de lecture, la classification de chaque question.
6. **Totaux :** À la fin, imprimer le nombre de questions `OK` (une seule marque), `BRANCO` et `DUPLA_MARCACAO`.

7.14.3.2 📌 Contraintes Computationnelles

- **Comparaison stricte :** seules les valeurs supérieures à Th sont considérées comme des marques valides ; les valeurs exactement égales au seuil ne doivent pas être comptabilisées.
- **Lettres des alternatives :** l'indice 0 correspond à l'alternative `A`, l'indice 1 à l'alternative `B`, et ainsi de suite.
- **Marques multiples :** dès que deux alternatives ou plus dépassent le seuil, la classification doit être `DUPLA_MARCACAO`, indépendamment de leurs degrés de remplissage respectifs.

7.14.3.3 🧠 Fondement Théorique

Situation	Classification	Interprétation
Exactement une alternative au-dessus du seuil	Lettre de l'alternative	Réponse valide
Aucune alternative au-dessus du seuil	`BRANCO`	Question non répondue
Deux alternatives ou plus au-dessus du seuil	`DUPLA_MARCACAO`	Réponse ambiguë

Le seuil de remplissage contrôle la sensibilité de l'algorithme. Des valeurs très basses tendent à augmenter le nombre de `DUPLA_MARCACAO`, tandis que des valeurs très hautes peuvent augmenter la quantité de questions classées comme `BRANCO`.

7.14.3.4 📦 Spécification d'Entrée et de Sortie (VPL)

Entrée :

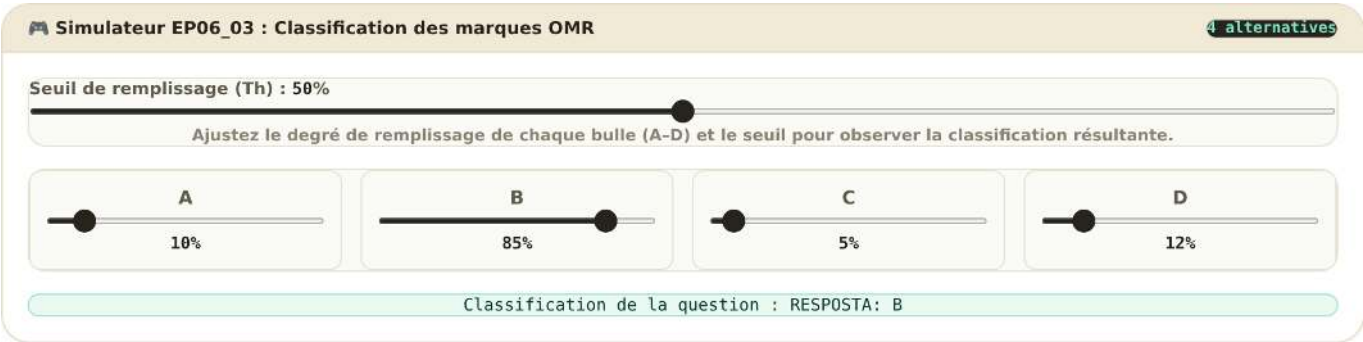
- Ligne 1 : Entier Q.
- Ligne 2 : Entier K.
- Ligne 3 : Nombre réel Th.
- Les Q lignes suivantes : K nombres réels, correspondant aux degrés de remplissage des alternatives.
- Ligne 1 : Entiers Q et K.

Sortie :

- Q lignes, chacune contenant la classification de la question respective.
- Ligne finale : `OK: x  BRANCO: y  DUPLA_MARCACAO: z`.

7.14.3.5 📌 Exemples

Entrée	Sortie	Observation
3 4 50 10 85 5 12 20 15 18 22 90 88 10 5	B BRANCO DUPLA_MARCACAO OK: 1 BRANCO: 1 DUPLA_MARCACAO: 1	Dans la première question, seule `B` dépasse le seuil ; dans la deuxième, aucune alternative ne le dépasse ; dans la troisième, `A` et `B` dépassent le seuil.
1 2 50.0 50 50	BRANCO OK: 0 BRANCO: 1 DUPLA_MARCACAO: 0	Les valeurs égales au seuil ne sont pas considérées comme des marques valides.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap06/sim-ep0603.html

Figure 7.23: Simulateur EP06_03 : Classification des marquages OMR

```
%%writefile EP06_03.py
# Code Python
```

```
Overwriting EP06_03.py
```

```
TestSuite("EP06_03.py").run()
```

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

7.14.4 EP06_04 🟡 Estimateur d'Inclinaison par Médiane Angulaire (*Deskew*)

Après la détection des contours et l'application de la Transformée de Hough, on obtient un ensemble de droites candidates pour l'orientation prédominante du document. Chaque droite fournit une estimation de l'angle d'inclinaison, calculée par

$$\text{angle} = \text{rad2deg}(\theta) - 90.$$

Cependant, toutes les droites ne correspondent pas aux lignes du document : certaines résultent de bruits, d'ombres ou d'autres éléments de l'image. Dans cet exercice, vous implémenterez l'étape d'estimation robuste de l'angle d'inclinaison, en filtrant les valeurs plausibles et en calculant leur médiane.

7.14.4.1 📋 Directives d'Implémentation

1. **Quantité :** Lire l'entier M, correspondant au nombre d'angles estimés.
2. **Angles :** Lire les M valeurs réelles, en degrés.
3. **Filtrage :** Conserver uniquement les angles qui satisfont **strictement** $-45 < \text{angle} < 45$.
4. **Absence de candidats :** Si aucun angle ne subsiste après le filtrage, imprimer exactement `SEM_CORRECAO`.
5. **Médiane :** S'il existe des angles valides :
   - si la quantité est impaire, la médiane est l'élément central de la séquence ordonnée ;
   - si elle est paire, la médiane est la moyenne arithmétique des deux éléments centraux.
6. **Sortie :** Imprimer la médiane arrondie à deux décimales (arrondi standard, *round half away from zero*, avec `np.floor(img + 0.5)`).

7.14.4.2 📌 Contraintes Computationnelles

- **Intervalle ouvert :** les angles égaux à -45 ou 45 ne doivent pas être considérés.
- **Précision :** calculer la médiane en utilisant les valeurs originales ; l'arrondi doit être effectué uniquement en sortie.
- **Cas vide :** s'il n'y a pas d'angles valides, aucune médiane ne doit être calculée.

7.14.4.3 🧠 Fondement Théorique

Situation	Résultat
Majorité des angles concentrée autour de l'inclinaison réelle	La médiane approxime l'orientation du document.
Peu d'angles aberrants (*outliers*)	La médiane subit peu d'influence de ces valeurs.
Angles hors de l'intervalle $(-45°, 45°)$	Ils sont écartés avant le calcul.
Aucun angle valide	Aucune correction n'est appliquée (`SEM_CORRECAO`).

La médiane est utilisée car elle est plus robuste que la moyenne en présence de quelques valeurs aberrantes, produisant une estimation plus stable de l'inclinaison prédominante du document.

7.14.4.4 📦 Spécification d'Entrée et de Sortie (VPL)

Entrée :

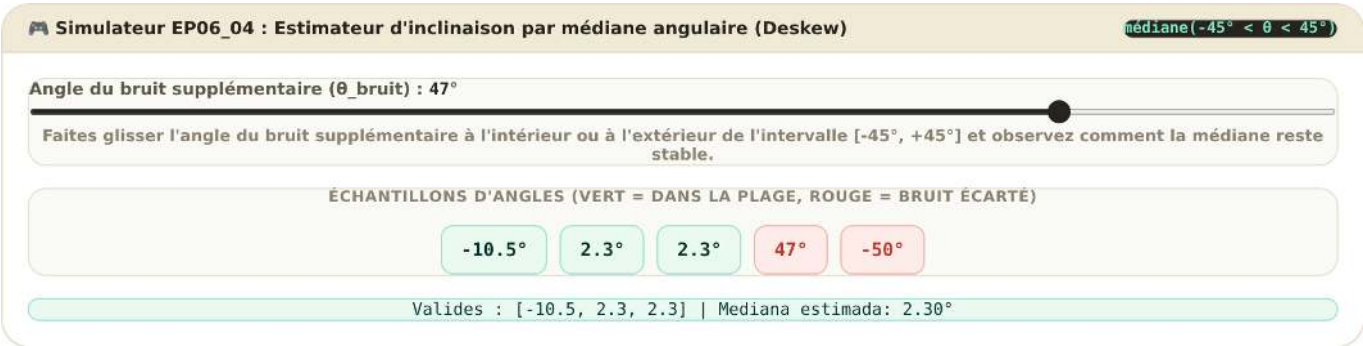
- Ligne 1 : Entier M.
- Ligne 2 : M nombres réels, correspondant aux angles en degrés.

Sortie :

- Une seule ligne contenant l'angle estimé, avec deux décimales, ou le mot `SEM_CORRECAO` si aucun angle n'est valide.

7.14.4.5 📌 Exemples

Entrée	Sortie	Observation
5 -50 -10.5 2.3 2.3 47	2.30	Seuls les angles dans l'intervalle $(-45, 45)$ sont considérés ; la médiane est 2,3.
4 -46 50 45 -45	SEM_CORRECAO	Aucun angle n'appartient à l'intervalle ouvert $(-45, 45)$.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap06/sim-ep0604.html

Figure 7.24: Simulateur EP06_04 : Estimateur de Pente par Médiane Angulaire

```
%%writefile EP06_04.py
# Code Python
```

```
Overwriting EP06_04.py
```

```
TestSuite("EP06_04.py").run()
```

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

7.14.5 EP06_05 🟠 Normalisation du fond par division (correction de l'éclairage)

Un formulaire a été photographié sous un éclairage non uniforme, ce qui fait qu'un côté de la feuille apparaît plus clair que l'autre. Dans ces conditions, le seuillage global par Otsu peut produire des résultats insatisfaisants, car un seul seuil ne sépare pas correctement le texte et le fond sur toute l'image. La solution présentée dans le chapitre consiste à **normaliser le fond**, en divisant l'image originale par une version fortement lissée d'elle-même, qui représente l'éclairage basse fréquence.

Dans cet exercice, l'image originale et le fond lissé (équivalent au résultat d'un `cv2.GaussianBlur` avec un σ élevé) sont déjà fournis. Votre tâche consiste à implémenter l'étape de normalisation qui produit l'image corrigée.

7.14.5.1 📋 Directives d'implémentation

1. **Dimensions :** Lire les entiers L (lignes) et C (colonnes).
2. **Image originale :** Lire les $L \times C$ valeurs entières de la matrice `img` (intensités entre 0 et 255).
3. **Fond estimé :** Lire les $L \times C$ valeurs entières de la matrice `bg` (intensités entre 0 et 255, toujours strictement supérieures à zéro).
4. **Normalisation :** Pour chaque position (i, j), calculer

$$\text{valeur}(i, j) = \frac{\text{img}(i, j)}{\text{bg}(i, j)} \times 255.$$

5. **Arrondi :** Arrondir le résultat à l'entier le plus proche (*round half away from zero*, avec `np.floor(img + 0.5)`).
6. **Saturation :** Limiter la valeur obtenue à l'intervalle $[0, 255]$.
7. **Sortie :** Imprimer la matrice `img_norm` résultante.

7.14.5.2 📌 Contraintes de calcul

- **Division par zéro :** l'entrée garantit $\text{bg}(i, j) > 0$ à toutes les positions.
- **Ordre des opérations :** d'abord arrondir, puis appliquer la saturation.
- **Traitement indépendant :** chaque pixel doit être normalisé individuellement, sans utiliser les informations des pixels voisins.

7.14.5.3 🧠 Fondement théorique

Situation	Effet de la normalisation
$\text{img}(i,j) = \text{bg}(i,j)$	Résultat égal à 255, correspondant au fond normalisé.
$\text{img}(i,j) < \text{bg}(i,j)$	Résultat inférieur à 255, préservant les régions plus sombres, comme le texte.
$\text{img}(i,j) > \text{bg}(i,j)$	Résultat supérieur à 255, ensuite saturé.
Fond avec éclairage non uniforme	La division réduit les variations lentes de l'éclairage, rendant l'image plus homogène.

La division par le fond estimé réduit les effets de l'éclairage non uniforme et préserve le contraste entre le premier plan et le fond, facilitant les étapes ultérieures de segmentation.

7.14.5.4 📦 Spécification des entrées et sorties (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Les L lignes suivantes : éléments de la matrice `img`.
- Les L lignes suivantes : éléments de la matrice `bg`.

Sortie :

- Matrice `img_norm`, avec L lignes et C colonnes, contenant des valeurs entières séparées par des espaces.

7.14.5.5 📌 Exemples

Entrée	Sortie	Observation
2 2 60 120 180 40 100 100 200 80	153 255 230 128	Les valeurs supérieures à 255 doivent être saturées ; $180/200 \times 255 = 229{,}5$ donne 230 après arrondi.
1 3 30 60 90 60 60 60	128 255 255	Seule la première valeur reste inférieure à 255 après la normalisation.

```
%%writefile EP06_05.py
# Code 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.fr/cap06/sim-ep0605.html

Figure 7.25: Simulateur EP06_05: Normalisation du fond par division

7.14.6 EP06_06 🔴 Carte de Variance Locale pour la Détection de Texture

Une usine de tissus doit inspecter des rouleaux de tissu en temps réel, sans disposer d'une image de référence — chaque rouleau présente de petites variations naturelles. Dans cette situation, la stratégie présentée dans le chapitre consiste à analyser l'**homogénéité locale de la texture** : les régions uniformes présentent une faible variance d'intensité dans de petits voisinages, tandis que les rayures, taches et défauts de fabrication produisent des augmentations locales de cette variance.

Dans cet exercice, vous implémenterez le cœur de cette méthode, en calculant la variance locale dans une fenêtre glissante et en générant un masque binaire qui identifie les régions dont la variance dépasse un seuil.

7.14.6.1 📋 Directives d'Implémentation

1. **Dimensions et paramètres :** Lire les entiers L, C, k (taille de la fenêtre, toujours impaire) et T (seuil de variance).
2. **Image :** Lire les $L \times C$ valeurs entières de la matrice de texture (intensités entre 0 et 255).
3. **Traitement des bords :** Lorsque la fenêtre dépasse les limites de l'image, utiliser la **réplication de bord**, c'est-à-dire répéter la valeur du pixel valide le plus proche.
4. **Moyenne locale :** Pour chaque position (i, j), calculer
$$\mu(i,j) = \frac{1}{k^2} \sum_{(p,q)\in\text{fenêtre}} \text{texture}(p,q).$$
5. **Variance locale :** Calculer la variance populationnelle de la fenêtre,
$$\sigma^2(i,j) = \frac{1}{k^2} \sum_{(p,q)\in\text{fenêtre}} \left(\text{texture}(p,q) - \mu(i,j)\right)^2,$$
ou, de manière équivalente,
$$\sigma^2(i,j) = \overline{x^2} - \mu(i,j)^2,$$
où $\overline{x^2}$ représente la moyenne des carrés des intensités.
6. **Arrondi :** Arrondir la variance à l'entier le plus proche (*round half away from zero*, avec `np.floor(res_norm + 0.5)`).
7. **Seuillage :** Définir masque$(i, j) = 1$ si la variance arrondie est **strictement supérieure** à T ; sinon, définir masque$(i, j) = 0$.
8. **Sortie :** Imprimer le masque binaire résultant.

7.14.6.2 📌 Contraintes Computationnelles

- **Réplication de bord :** utiliser la valeur du pixel valide le plus proche chaque fois que la fenêtre dépasse les limites de l'image.
- **Variance populationnelle :** utiliser le dénominateur k^2, jamais $k^2 - 1$.
- **Comparaison stricte :** le masque doit être calculé en utilisant la condition $\sigma^2_{\text{arrondi}} > T$.
- **Fenêtre impaire :** la valeur de k est toujours impaire, garantissant un pixel central.

7.14.6.3 🧠 Fondement Théorique

Situation	Variance locale	Interprétation
Région uniforme	Faible	Intensités similaires dans le voisinage.
Région contenant un défaut	Élevée	La présence d'intensités distinctes augmente la dispersion des valeurs.
Petite fenêtre	Plus grande sensibilité aux détails et au bruit	Détecte les altérations localisées.
Grande fenêtre	Réponse plus lisse	Met en évidence les défauts plus grands, mais réduit la précision de leur localisation.

La variance locale mesure la dispersion des intensités dans un voisinage. Les régions homogènes présentent une faible variance, tandis que les altérations de la texture augmentent cette mesure, permettant d'identifier d'éventuels défauts par un simple seuillage.

7.14.6.4 📦 Spécification d'Entrée et de Sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Entier k (impair).
- Ligne 4 : Entier T.
- Lignes suivantes L : éléments entiers de la matrice de texture.

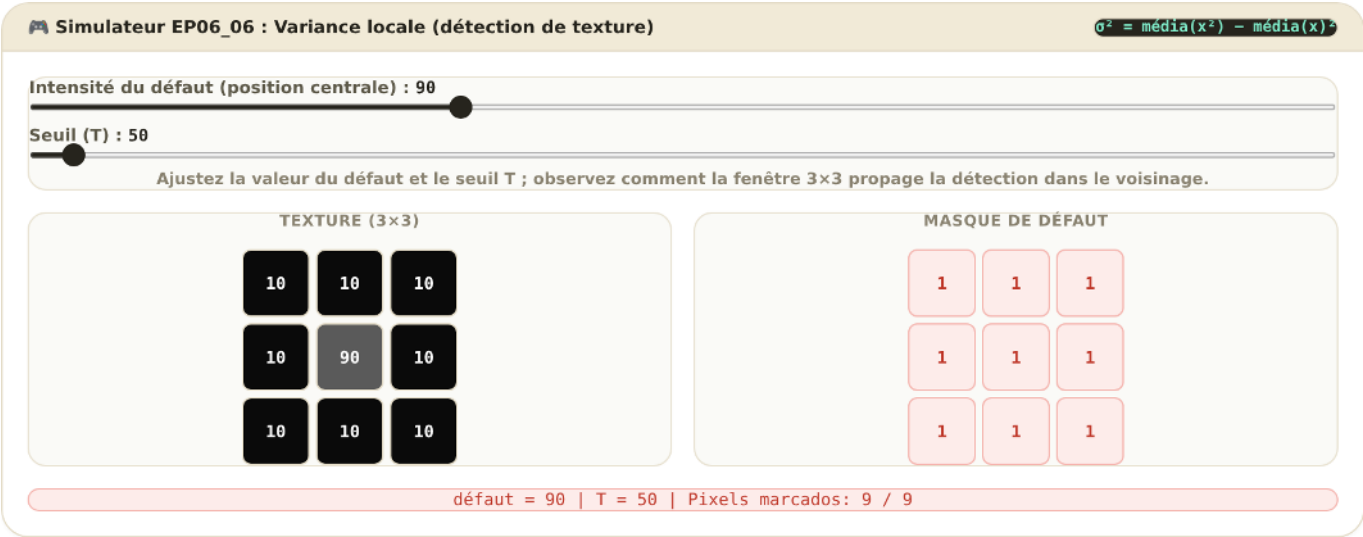
Sortie :

- Masque binaire (valeurs 0 ou 1), avec L lignes et C colonnes.

7.14.6.5 📌 Exemples

Entrée	Sortie	Observation
3 3 3 50 10 10 10 10 10 10 10 90 10	0 0 0 1 1 1 1 1 1	Le défaut augmente la variance dans toutes les fenêtres qui le contiennent.

Entrée	Sortie	Observation
2 2 3 5 100 100 100 100	0 0 0 0	La texture est uniforme ; la variance est nulle sur toute l'image.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap06/sim-ep0606.html

Figure 7.26: Simulateur EP06_06 : Carte de variance locale pour la détection de texture

```
%%writefile EP06_06.py
# Code Python
```

```
Overwriting EP06_06.py
```

```
TestSuite("EP06_06.py").run()
```

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

7.14.7 EP06_07 🟣 *Pipeline* d'inspection industrielle : enregistrement par translation et soustraction

Sur une chaîne de production, une caméra fixe photographie chaque pièce passant sur le tapis roulant, en la comparant à une image de référence sans défauts. Le problème : de petites vibrations du tapis déplacent la pièce par rapport à la position de référence à chaque capture. Si la soustraction d'images est appliquée directement, sans correction, le déplacement en lui-même génère déjà d'énormes différences — **faux positifs** qui masquent les défauts réels.

Cet exercice est le plus complet du chapitre : vous devez **d'abord enregistrer** (aligner géométriquement) l'image capturée en utilisant un déplacement connu (dx, dy), fourni par un capteur de position du tapis, et **seulement ensuite appliquer la soustraction** avec seuillage, exactement comme décrit dans la section sur l'inspection industrielle.

7.14.7.1 📋 Directives d'implémentation

1. **Dimensions et paramètres :** Lire L, C (dimensions des images), le déplacement entier connu dx, dy (pouvant être négatifs) et le seuil de détection T (entier).

2. **Images :** Lire la matrice de référence (`ref`, $L \times C$, sans défauts) et la matrice capturée (`cap`, $L \times C$, éventuellement déplacée et avec défaut).
3. **Enregistrement par translation :** Construire l'image alignée `alin` en appliquant le déplacement (dx, dy) reçu :

$$\text{alin}(i,j) = \begin{cases} \text{cap}(i+dy,\ j+dx), & \text{si } (i+dy,\ j+dx) \in [0,L) \times [0,C) \\ 0, & \text{sinon} \end{cases}$$

4. **Remplissage des bords :** Les positions qui « sortent » de l'image capturée après le déplacement reçoivent la valeur **0** (*zero-padding* — hors du champ de vision de la caméra ; **notez que cet exercice utilise zéro, contrairement à la réplication des bords de l'EP06_06**).
5. **Différence absolue :** Calculer, pixel par pixel,

$$\text{diff}(i,j) = |\text{ref}(i,j) - \text{alin}(i,j)|$$

6. **Seuillage :** Définir $\text{masque}(i,j) = 1$ si $\text{diff}(i,j) > T$; sinon, $\text{masque}(i,j) = 0$.
7. **Sortie :** Dans cet ordre — (a) la matrice `alin` ($L \times C$) ; (b) le masque de défaut ($L \times C$) ; (c) une dernière ligne avec le nombre total de pixels classés comme défectueux.

7.14.7.2 📌 Contraintes computationnelles

- ***Zero-padding*, pas de réplication :** les positions hors des limites de l'image capturée, après le déplacement, valent exactement 0 — c'est le point qui différencie le plus cet exercice de l'EP06_06.
- **Comparaison stricte :** $\text{diff}(i,j) > T$.
- **Signe de (dx, dy) :** le déplacement peut être positif ou négatif ; la formule de l'étape 3 doit être appliquée littéralement, sans inverser les signes.
- **Toutes les valeurs sont des entiers :** aucun arrondi n'est effectué à cette étape.

7.14.7.3 🧠 Fondement théorique

Étape omise	Conséquence
Sauter l'enregistrement géométrique	Le bord entier de l'image (introduit par le déplacement) est marqué comme « défaut » — faux positif systématique
Enregistrement avec (dx, dy) incorrect	La pièce et la référence restent désalignées ; la soustraction détecte des contours déplacés, pas de véritables défauts
Seuil T trop bas	Le bruit de capture (variations de 1 à 2 niveaux de gris) est confondu avec un défaut
Seuil T trop élevé	Les défauts subtils ne sont plus détectés

L'enregistrement géométrique et la soustraction sont des étapes complémentaires : le premier garantit que les deux images représentent exactement la même scène dans le même référentiel spatial ; la seconde isole ce qui a réellement changé entre elles — idéalement, uniquement les défauts.

7.14.7.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- Ligne 2 : Entier C.
- Ligne 3 : Deux entiers dx et dy, séparés par un espace.
- Ligne 4 : Entier T.
- Les L lignes suivantes : éléments entiers de la matrice `ref`.

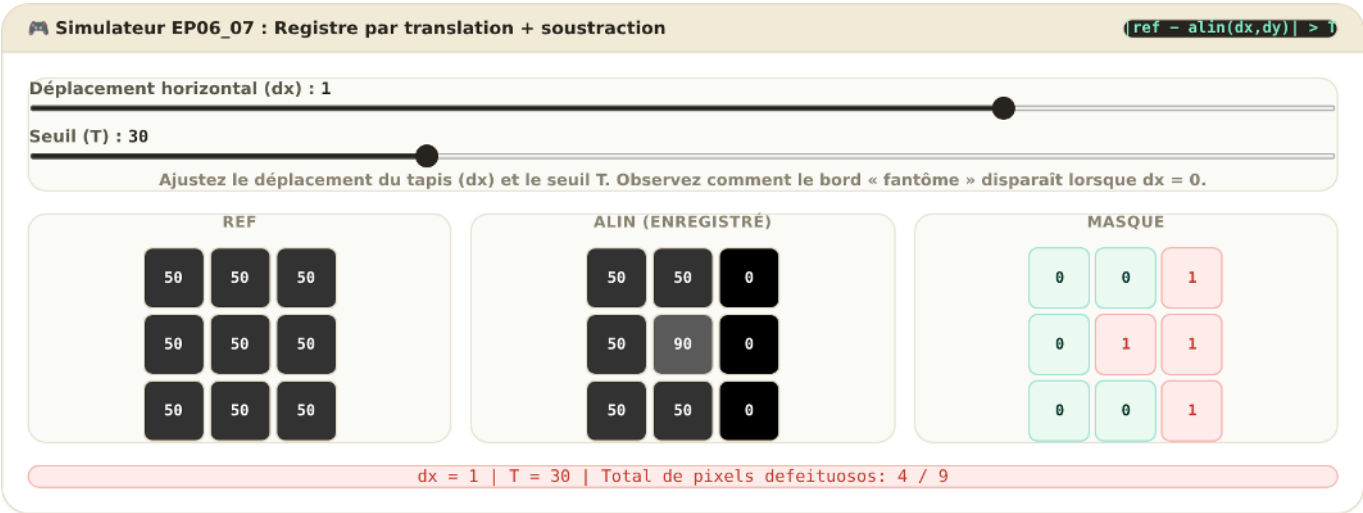
- Les L lignes suivantes : éléments entiers de la matrice `cap`.

Sortie :

- L lignes avec la matrice `alin`.
- L lignes avec le masque de défaut (0/1).
- Dernière ligne : `Total de pixels défectueux : X`.

7.14.7.5 📌 Exemples

Entrée	Sortie	Observation
3 3 1 0 30 50 50 50 50 50 50 50 50 50 0 50 50 0 50 90 0 50 50	50 50 0 50 90 0 50 50 0 0 0 1 0 1 1 0 0 1 Total de pixels défectueux : 4	$dx = 1$ décale la lecture d'une colonne vers la droite ; la dernière colonne de `alin` n'a pas de correspondance (devient 0) et est systématiquement marquée ; le défaut réel (90) est également détecté.
2 2 0 0 20 10 10 10 10 10 10 10 60	10 10 10 60 0 0 0 1 Total de pixels défectueux : 1	Sans déplacement ($dx = dy = 0$) : `alin` est identique à `cap` ; seul le défaut réel (60) est détecté.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap06/sim-ep0607.html

Figure 7.27: Simulateur EP06_07 : *Pipeline* d'inspection — Enregistrement par translation et soustraction

```
%%writefile EP06_07.py
# Code Python
```

```
Overwriting EP06_07.py
```

```
TestSuite("EP06_07.py").run()
```

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

7.14.8 EP06_08 ⚫ Segmentation et Décodage Réel de *QRCode* avec OpenCV

Dans les exercices précédents, les grandeurs intermédiaires du *pipeline* de traitement d'images — telles que les aires, les périmètres, les variances et les déplacements — étaient fournies directement ou calculées à partir de matrices numériques, sans nécessiter de bibliothèques spécialisées de Vision par Ordinateur. Dans cet exercice de clôture du chapitre, cette restriction est levée de manière intentionnelle : la bibliothèque **OpenCV** (`cv2`) sera utilisée pour localiser et décoder un *QRCode* réel présent dans une scène.

La proposition reproduit un flux simplifié de systèmes employés en inspection visuelle, en automatisation industrielle et en lecture automatique de documents. Pour maintenir l'entrée de données accessible au contexte éducatif, le chargement de l'image sera intégré à la bibliothèque didactique `morph`, via la fonction `mm.readImg`.

La scène est fournie au format **PGM ASCII (P2)** et contient un unique *QRCode* valide, ainsi que divers **objets distracteurs**, tels que des rectangles, des zones de bruit texturé et des blocs isolés. La segmentation basée uniquement sur des propriétés géométriques — comme l'aire et la forme approximativement carrée — est nécessaire pour réduire l'espace de recherche, mais ne suffit pas à identifier le code correct. La confirmation finale sera réalisée exclusivement par la tentative de décodage avec `cv2.QRCodeDetector`, procédure compatible avec les applications réelles de reconnaissance automatique.

7.14.8.1 📋 Directives d'Implémentation

1. **Lecture des dimensions et des paramètres**

   Lire, dans cet ordre, via l'entrée standard :

   - une ligne contenant le nombre de lignes L ;
   - une ligne contenant le nombre de colonnes C ;
   - une ligne contenant les quatre paramètres de l'algorithme séparés par des espaces :
     - le seuil de binarisation T (entier) ;
     - l'aire minimale A_{min} (entier) ;
     - la tolérance d'aspect tol (réel) ;
     - la marge M (entier, en pixels).

2. **Chargement de l'image**

   Utiliser la fonction didactique `f = mm.readImg(L, C)` pour lire les $L \times C$ valeurs de l'image en niveaux de gris, obtenant ainsi un *array* NumPy de type `uint8`.

3. **Binarisation**

   Appliquer une binarisation inversée en utilisant le seuil T. Tout pixel de l'image originale avec une intensité strictement supérieure à T doit être converti à 255, tandis que les autres doivent prendre la valeur 0.

4. **Détection des contours**

   Extraire les composants connectés externes en utilisant `cv2.findContours(...)` avec les paramètres :

   - `cv2.RETR_EXTERNAL` ;
   - `cv2.CHAIN_APPROX_SIMPLE`.

5. **Filtrage géométrique**

   Pour chaque contour trouvé :

   - calculer le rectangle englobant `(x, y, w, h)` à l'aide de `cv2.boundingRect` ;

- ne conserver que les candidats satisfaisant simultanément :

  Aire minimale

  $$w \times h > A_{\min}$$

  Rapport d'aspect

  $$\left|\frac{w}{h} - 1\right| \leq \text{tol}$$

6. **Tri des candidats**

   Trier les candidats par l'aire du rectangle englobant

   $$w \times h$$

   en ordre décroissant.

   En cas d'égalité, préserver l'ordre originalement retourné par `cv2.findContours`.

7. **Vérification par décodage**

   Pour chaque candidat, en suivant l'ordre établi :

   - étendre le rectangle de M pixels dans les quatre directions ;
   - limiter les indices pour rester dans l'image ;
   - extraire le recadrage directement depuis l'image originale `f` ;
   - appliquer `cv2.QRCodeDetector().detectAndDecode(...)` sur ce recadrage.

8. **Critère d'arrêt**

   Interrompre immédiatement le traitement lorsque le premier candidat produit une *chaîne* décodée non vide.

9. **Cas non trouvé**

   Si aucun candidat n'est décodé avec succès, imprimer exactement : `QRCODE_NAO_ENCONTRADO`

10. **Sortie (cas trouvé)**

    Imprimer deux lignes.

    Première ligne : `linha coluna altura largura` en utilisant le rectangle englobant **original**, avant l'expansion par la marge M.

    Deuxième ligne : `texto_decodificado`

7.14.8.2 📌 Contraintes Computationnelles

- Utiliser les fonctions d'OpenCV pour réaliser la binarisation, la détection des contours, le calcul du rectangle englobant et le décodage du QRCode.
- Le filtrage géométrique doit obligatoirement avoir lieu avant l'étape de décodage.
- Utiliser exclusivement le seuil fixe T fourni en entrée. Il n'est pas permis d'utiliser des méthodes automatiques de seuillage, comme Otsu ou le seuillage adaptatif.
- Garantir que les recadrages envoyés au décodeur restent dans les limites de l'image.

7.14.8.3 🧠 Fondement Théorique

Étape	Rôle dans le pipeline	Conséquence si omise
Filtrage géométrique	Réduit l'espace de recherche en sélectionnant uniquement les régions compatibles avec la géométrie attendue d'un QRCode.	Le décodeur traiterait tous les contours, y compris les bruits et les objets distracteurs.
Décodage	Confirme sémantiquement si le candidat contient un QRCode valide.	Des objets géométriquement similaires pourraient être classifiés incorrectement comme QRCode.
Marge M	Préserve la *zone de silence* autour du code, facilitant sa détection.	L'absence de cette marge peut empêcher l'alignement et la lecture correcte du code.

Cet exercice intègre les concepts étudiés tout au long du chapitre dans un *pipeline* unique de Vision par Ordinateur. La segmentation réduit l'ensemble des régions candidates grâce à des caractéristiques géométriques, tandis que l'étape de décodage valide le contenu de la région en utilisant un algorithme spécialisé de reconnaissance.

7.14.8.4 📦 Spécification d'Entrée et de Sortie (VPL)

Structure d'Entrée

```
L
C
T A_min tol M
[matrice de l'image]
```

Structure de Sortie (Succès)

```
linha coluna altura largura
texto_decodificado
```

Structure de Sortie (Échec)

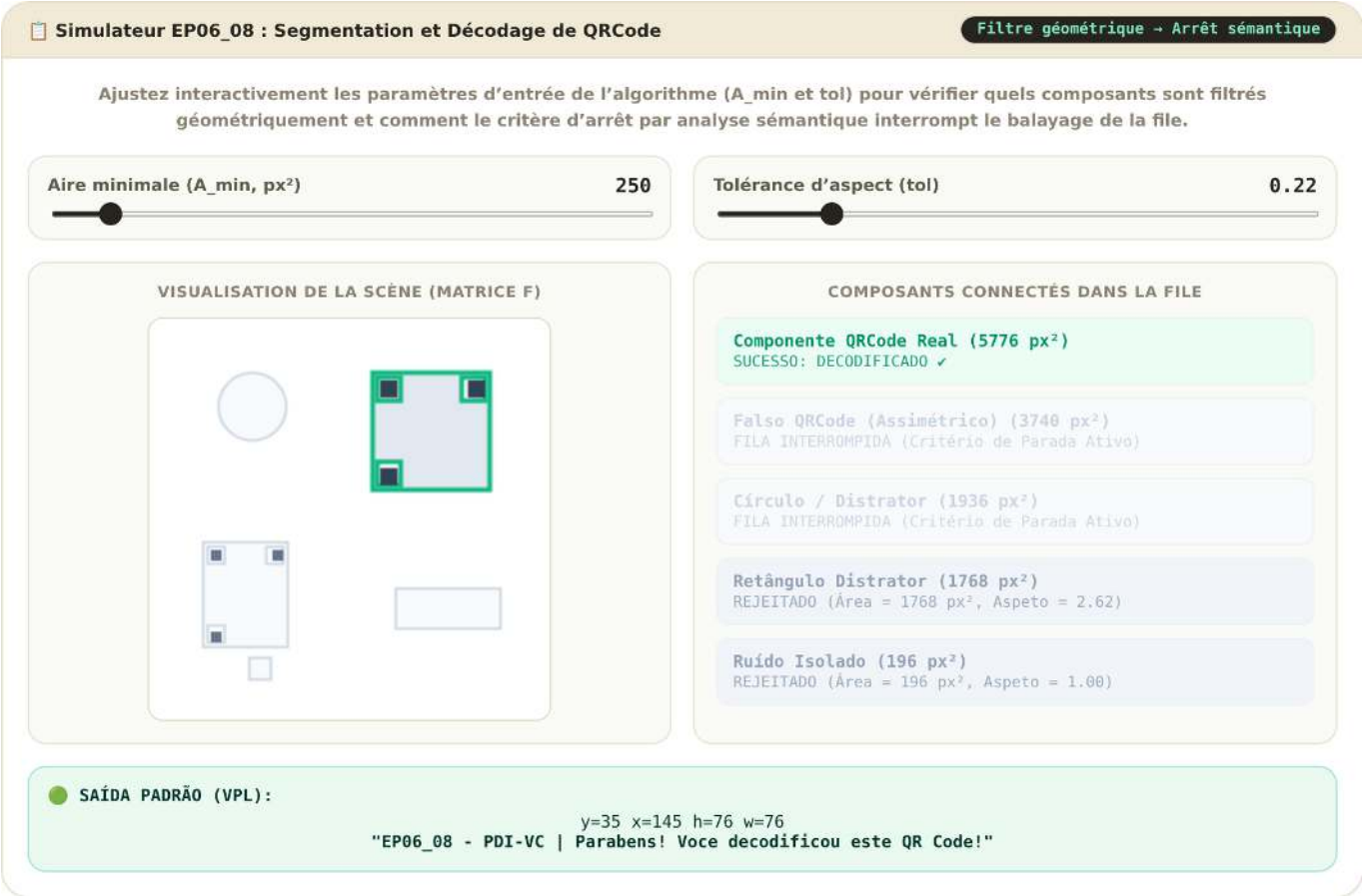
```
QRCODE_NAO_ENCONTRADO
```

7.14.8.5 📌 Fichiers de Référence (.pgm)

À des fins de validation, de débogage local et d'analyse de matrices réelles de pixels, les fichiers d'image générés au format ASCII P2 sont disponibles dans le répertoire du projet. Vous pouvez les utiliser pour tester la conformité de votre code en les décodant avec votre téléphone (sauvegardez localement les *.pgm pour les visualiser) :

- 📥 **Cas 1 : Schéma Normal** – Contient un unique code parfaitement centré avec des distracteurs géométriques simples en périphérie.
- 📥 **Cas 2 : Scénario Complexe** – Présente une densité de bruit texturé plus élevée et de multiples candidats distracteurs qui testent les limites du filtrage par aspect.
- 📥 **Cas 3 : Message Étendu** – Contient un QRCode structuré à partir d'une chaîne de caractères plus longue, générant une densité de modules internes plus élevée.
- 📥 **Cas 4 : Géométrie Compacte** – Évalue le comportement du pipeline dans des conditions optimisées de contraste et de positionnement limite.
- 📥 **Cas 5 : Scénario d'Exclusion** – Image composée purement d'éléments distracteurs de grande aire, conçue pour valider le comportement de défaillance contrôlée du programme.

```
%%writefile EP06_08.py
# Code Python
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap06/sim-ep0608.html

Figure 7.28: Simulateur EP06_08 : Segmentation géométrique + Vérification par décodage de QRCode

```
Overwriting EP06_08.py
```

```
TestSuite("EP06_08.py").run()
```

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

Chapitre 8

Classification d'images et reconnaissance de formes

Exécuter Colab Ouvrir GitHub

Dans le **chapitre 6**, la transition de la partie I vers la partie II a été présentée à travers deux applications qui exigeaient déjà des décisions automatisées : la reconnaissance de marques sur des feuilles de réponses (OMR) et la détection de défauts en inspection industrielle. Dans les deux cas, cependant, les décisions dépendaient de règles géométriques et de seuils définis manuellement, comme déterminer si un disque était suffisamment circulaire ou si une région était suffisamment sombre.

Ce chapitre formalise le problème plus général sous-jacent à ces applications : étant donné un ensemble d'exemples étiquetés, comment entraîner un système pour **classifier automatiquement** de nouvelles images ou régions d'intérêt ? Cette question est au cœur de la **reconnaissance de formes**, discipline qui fonde une grande partie des tâches modernes de vision par ordinateur, depuis la classification d'images jusqu'à la détection d'objets et la segmentation sémantique, explorées dans les chapitres suivants.

Seront étudiés les principaux **descripteurs classiques d'image** (couleur, texture et forme/gradient) ainsi que le classifieur **k-plus proches voisins** (*k-Nearest Neighbors* — k-NN), choisi pour sa simplicité conceptuelle et pour mettre en évidence, de manière directe, la relation entre l'espace des caractéristiques, les métriques de distance et les frontières de décision — des concepts qui demeurent centraux même dans les classifieurs fondés sur les réseaux de neurones profonds, étudiés dans le chapitre final de cette partie.

8.1 Objectifs du chapitre

À la fin de ce chapitre, l'étudiant devrait être capable de :

- **Comprendre le *pipeline* classique de reconnaissance de formes** : acquisition, prétraitement, extraction de descripteurs, classification et évaluation ;
- **Extraire et interpréter des descripteurs classiques** de couleur, de texture (*Local Binary Patterns* — LBP) et de forme/gradient (*Histogram of Oriented Gradients* — HOG) ;
- **Implémenter et entraîner un classifieur k-NN** pour des tâches de classification d'images ;
- **Évaluer les classifieurs** à l'aide de métriques telles que l'exactitude, la matrice de confusion, la précision et le rappel ;
- **Analyser l'effet du paramètre k** et de la dimensionnalité de l'espace des caractéristiques sur la performance du classifieur ;
- **Reconnaître les limites des descripteurs artisanaux** (*hand-crafted features*) et comprendre la motivation pour la transition, dans les chapitres suivants, vers des descripteurs appris automatiquement.

8.2 Configuration de l'environnement

Les exemples de ce chapitre utilisent des bibliothèques largement employées en traitement numérique des images, en vision par ordinateur et en apprentissage automatique. Le bloc ci-dessous installe les paquets nécessaires ; dans les environnements où ils sont déjà présents, l'exécution peut être ignorée.

```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 importlib
import subprocess
import sys

def setup_cap07():
    """Installe les bibliothèques manquantes nécessaires pour ce chapitre
    (vision par ordinateur et apprentissage automatique)."""
    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"[AVERTISSEMENT] Échec de l'installation de {pacote} (nécessaire pour
                le module {modulo}).")


setup_cap07()

# ==========================================================
# Bibliothèques
# ==========================================================

# Calcul scientifique
import numpy as np
import pandas as pd

# Visualisation
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import seaborn as sns

# Vision par ordinateur
import cv2
from skimage import data as skdata
from skimage.feature import hog, local_binary_pattern

# Apprentissage automatique
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,
)
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0
```

8.3 Un Problème Concret : Classification des Fruits

Avant de présenter les fondements théoriques, considérons le problème suivant, qui servira d'exemple tout au long de ce chapitre pour illustrer les principaux concepts de la reconnaissance de formes.

Le scénario : Une ferme automatisée utilise un système de vision par ordinateur pour séparer les pommes, les bananes et les oranges sur des lignes d'emballage.

Le défi : Les fruits arrivent sur le tapis roulant dans différentes positions et orientations, sous des conditions d'éclairage qui peuvent varier. De plus, les feuilles, les ombres et les petites occlusions peuvent rendre leur identification difficile. Comment développer un système capable de les classer correctement ?

Une approche possible :

1. Extraire des descripteurs qui représentent des caractéristiques pertinentes des fruits :
   - **Couleur :** distribution prédominante des couleurs ;
   - **Texture :** différences à la surface de la peau ;
   - **Forme :** caractéristiques géométriques du contour.
2. Entraîner un classificateur en utilisant des exemples préalablement étiquetés.
3. Utiliser le modèle entraîné pour classer automatiquement de nouveaux fruits.

La Figure 8.1 illustre, de manière conceptuelle, comment différentes fruits peuvent être représentés dans un espace de caractéristiques tridimensionnel.

> **Réfléchissez avant de continuer**
>
> Si chaque fruit était représenté uniquement par les valeurs d'intensité de ses pixels, serait-il possible de les distinguer de manière fiable ? Quels types d'informations pourrait-on extraire de l'image pour faciliter cette tâche ?

```
np.random.seed(42)

centros = {
    "Maçã":    [0.8, 0.2, 0.9],
    "Banana":  [0.3, 0.1, 0.2],
    "Laranja": [0.9, 0.8, 0.8],
}

fig = plt.figure(figsize=(5, 5))
ax = fig.add_subplot(projection="3d")

for fruta, centro, cor, marcador in zip(
```

```
    centros,
    centros.values(),
    ["red", "gold", "orange"],
    ["o", "s", "^"],
):
    X = np.clip(np.random.normal(centro, 0.08, (70, 3)), 0, 1)
    ax.scatter(X[:, 0], X[:, 1], X[:, 2],
               c=cor, marker=marcador, s=35,
               alpha=0.7, label=fruta)

ax.set(
    xlim=(0,1), ylim=(0,1), zlim=(0,1),
    xlabel="Intensidade de cor",
    ylabel="Textura",
    zlabel="Forma",
    title="Espaço de Características"
)
ax.view_init(elev=25, azim=-60)
ax.legend(title="Frutas")
ax.zaxis.labelpad = 0.01

plt.tight_layout()
plt.show()
```

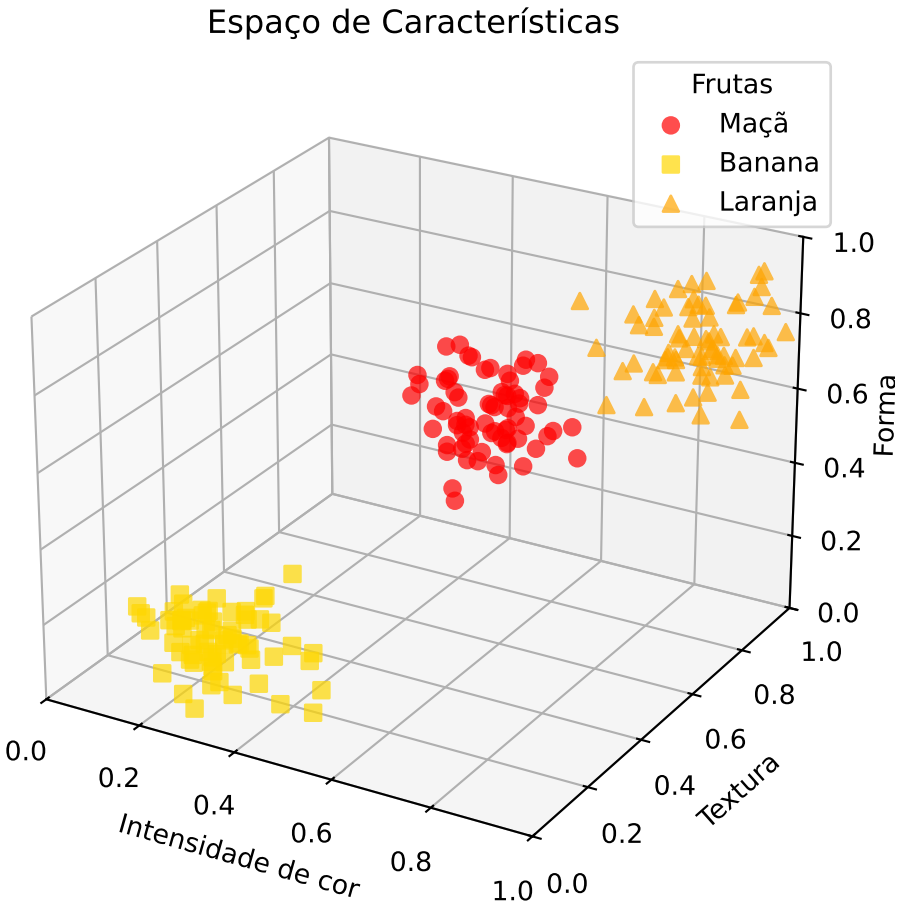


Figure 8.1: Exemple motivationnel : différents fruits formant des regroupements distincts dans un espace de caractéristiques.

8.4 Aperçu du Chapitre : Le *Pipeline* Classique de Classification d'Images

Avant de poursuivre, il est utile de présenter une vue d'ensemble intégrée de ce qui sera étudié. La Figure 9.1 illustre le flux général d'un système classique de classification d'images, depuis l'image d'entrée jusqu'à l'étape d'attribution du label final. Dans les sections suivantes, chacune des étapes de ce processus sera étudiée en détail.

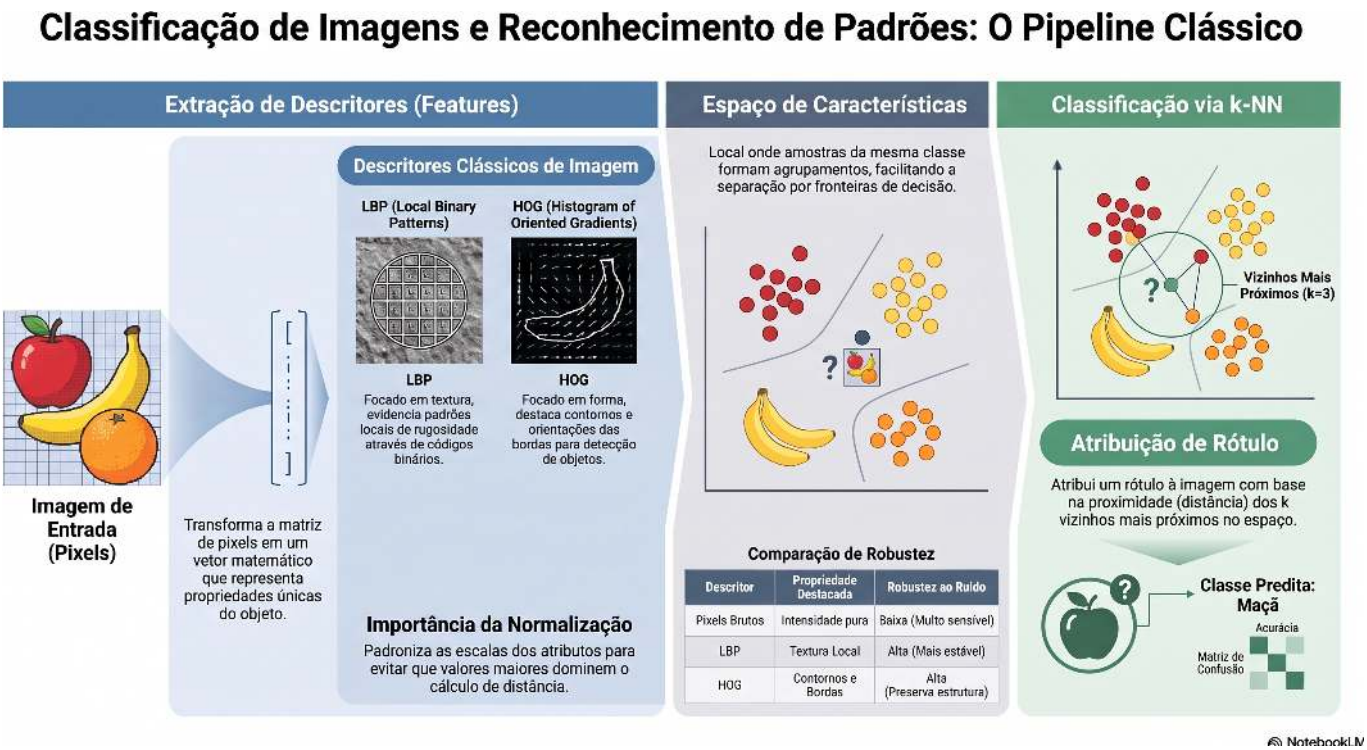


Figure 8.2: Aperçu du *pipeline* classique de classification d'images : extraction de descripteurs (LBP, HOG), formation de l'espace des caractéristiques et classification via k-NN. Ne s'applique pas aux modèles d'apprentissage profond (CNN, YOLO), qui apprennent des *features* de bout en bout directement à partir des pixels. **Source :** élaboré avec l'aide de Gemini Notebook ({GOOGLE}, 2025).

> **ℹ Portée de ce chapitre**
>
> Dans ce chapitre, l'accent est mis exclusivement sur le classificateur k-NN, en raison de sa simplicité pédagogique et de sa capacité à illustrer de manière intuitive le concept d'espace des caractéristiques. D'autres classificateurs traditionnels largement utilisés en reconnaissance de formes — tels que les arbres de décision, les règles de classification et les machines à vecteurs de support (SVM) — sont discutés en profondeur dans Quilici-gonzalez (2014), notamment dans sa 2e édition, actuellement en production (QUILICI-GONZALEZ, 2026).

8.5 Fondements de la reconnaissance de formes

Un système de **reconnaissance de formes** a pour objectif d'attribuer une catégorie (étiquette) à une observation — une image entière, une région d'intérêt ou un signal — sur la base d'exemples préalablement étiquetés. De manière générale, ce processus est organisé selon les étapes suivantes :

1. **Acquisition :** obtention de l'image ou du signal à classer ;
2. **Prétraitement :** normalisation, suppression du bruit, correction géométrique ou d'éclairage — étapes déjà étudiées dans les chapitres précédents ;

3. **Extraction de descripteurs (*features*) :** transformation de l'image en un **vecteur de caractéristiques** de dimension fixe, qui représente les propriétés pertinentes pour la tâche de classification ;
4. **Classification :** application d'un modèle qui associe le vecteur de caractéristiques à une classe ;
5. **Évaluation :** analyse des performances du modèle sur un ensemble de données indépendant de celui utilisé pour l'entraînement.

L'ensemble de tous les vecteurs de caractéristiques possibles constitue l'**espace de caractéristiques** (*feature space*). Un bon descripteur produit des représentations qui rapprochent, dans cet espace, les observations d'une même classe et éloignent les observations de classes distinctes. Cette propriété favorise les méthodes de classification basées sur la proximité, comme le **k-NN**, et bénéficie également à divers autres classifieurs.

La Figure 8.1 illustre ce concept de manière schématique : chaque fruit est représenté par un point dans un espace de caractéristiques à trois dimensions (couleur, texture et forme). Bien que cet espace ne soit qu'une simplification didactique, il montre comment les échantillons d'une même classe tendent à former des regroupements, tandis que des classes différentes occupent des régions distinctes, facilitant ainsi la tâche de classification.

8.6 Extraction de descripteurs classiques

Avant la popularisation des réseaux de neurones profonds, les descripteurs d'image étaient, pour la plupart, conçus manuellement par des experts (*hand-crafted features*), sur la base de propriétés statistiques ou géométriques connues. Trois familles classiques sont particulièrement pertinentes :

- **Descripteurs de couleur :** histogrammes d'intensité ou de teinte, qui capturent la distribution des valeurs de couleur d'une région, déjà introduits au **Chapitre 3** via la fonction `mm.hist` ;
- **Descripteurs de texture :** capturent des motifs locaux de répétition, de rugosité ou d'orientation, comme le *Local Binary Patterns* (LBP), étudié ci-après ;
- **Descripteurs de forme/gradient :** décrivent la distribution des contours et des orientations du gradient, comme le *Histogram of Oriented Gradients* (HOG), largement utilisé dans la détection de personnes et d'autres objets.

Pour comparer l'information capturée par chaque approche, la Figure 8.3 montre comment différentes techniques « voient » la même image.

```
# Charger une image d'exemple
imagem = skdata.camera()

# Appliquer les descripteurs
lbp_img = local_binary_pattern(
    imagem,
    P=8,
    R=1,
    method="uniform"
    )

# Convertir le LBP en RGB uniquement pour faciliter la visualisation
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,
)
```

```
# Affichage standardisé
mm.show(
    [imagem, lbp_rgb, hog_img],
    titles=[
        "Image originale\n(comme l'humain la voit)",
        "LBP : Texture\n(chaque couleur = un code LBP)",
        "HOG : Gradients et contours\n(zones claires = intensité plus élevée)",
    ],
    cols=3,
    figsize=(12, 4),
)

print("Observez comment chaque descripteur met en évidence des propriétés différentes :")
print("• LBP : met en évidence les motifs locaux de texture.")
print("• HOG : met en évidence les contours et les orientations des bords.")
print("• Image originale : contient uniquement les valeurs d'intensité.")
```

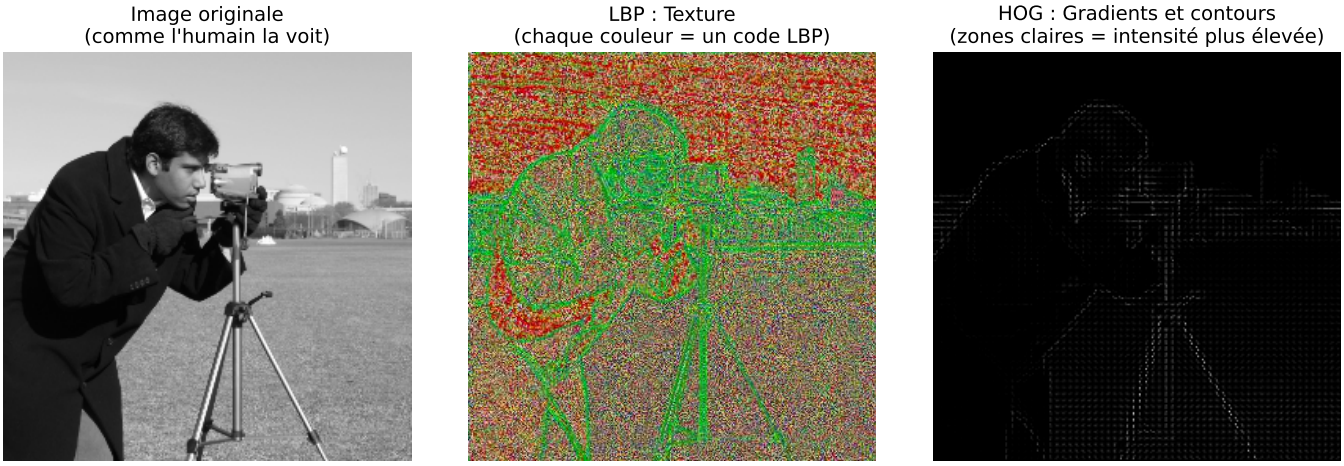


Figure 8.3: Comparaison visuelle de différents descripteurs appliqués à la même image. Chaque descripteur révèle des aspects distincts de la scène.

```
Observez comment chaque descripteur met en évidence des propriétés différentes :
• LBP : met en évidence les motifs locaux de texture.
• HOG : met en évidence les contours et les orientations des bords.
• Image originale : contient uniquement les valeurs d'intensité.
```

8.6.1 *Local Binary Patterns* (LBP)

Le LBP est un descripteur de texture qui code, pour chaque pixel central g_c, la relation entre son intensité et celle des P voisins disposés dans un voisinage circulaire de rayon R :

$$\mathrm{LBP}_{P,R}(x_c, y_c) = \sum_{p=0}^{P-1} s(g_p - g_c)\, 2^p, \qquad s(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

où :

- (x_c, y_c) sont les coordonnées du pixel central ;
- g_c est l'intensité du pixel central ;
- g_p est l'intensité du p-ième pixel voisin ;
- P est le nombre de voisins considérés ;
- R est le rayon du voisinage circulaire ;
- p est l'indice du voisin, avec $p = 0, 1, \ldots, P-1$;

- $s(z)$ est la fonction de seuillage définie dans l'équation, où $z = g_p - g_c$; elle prend la valeur 1 lorsque $z \geq 0$ et 0 lorsque $z < 0$;
- 2^p correspond au poids binaire associé au p-ième voisin.

Le code LBP obtenu décrit le motif local de contraste autour du pixel. L'histogramme de ces codes forme un vecteur de caractéristiques compact pour représenter la texture de l'image (Figure 8.4). Dans ce chapitre, on utilise la variante **uniforme**, qui regroupe les motifs non uniformes dans une seule catégorie, réduisant la dimensionnalité et augmentant la robustesse du descripteur.

```
plt.figure(figsize=(6, 4))

plt.hist(
    lbp_img.ravel(),
    bins=np.arange(-0.5, lbp_img.max() + 1.5, 1),
    density=True,
    edgecolor="black",
)

plt.title("Histograma dos códigos LBP")
plt.xlabel("Código LBP")
plt.ylabel("Frequência relativa")
plt.xticks(range(int(lbp_img.max()) + 1))
plt.grid(axis="y", alpha=0.3)

plt.tight_layout()
plt.show()
```

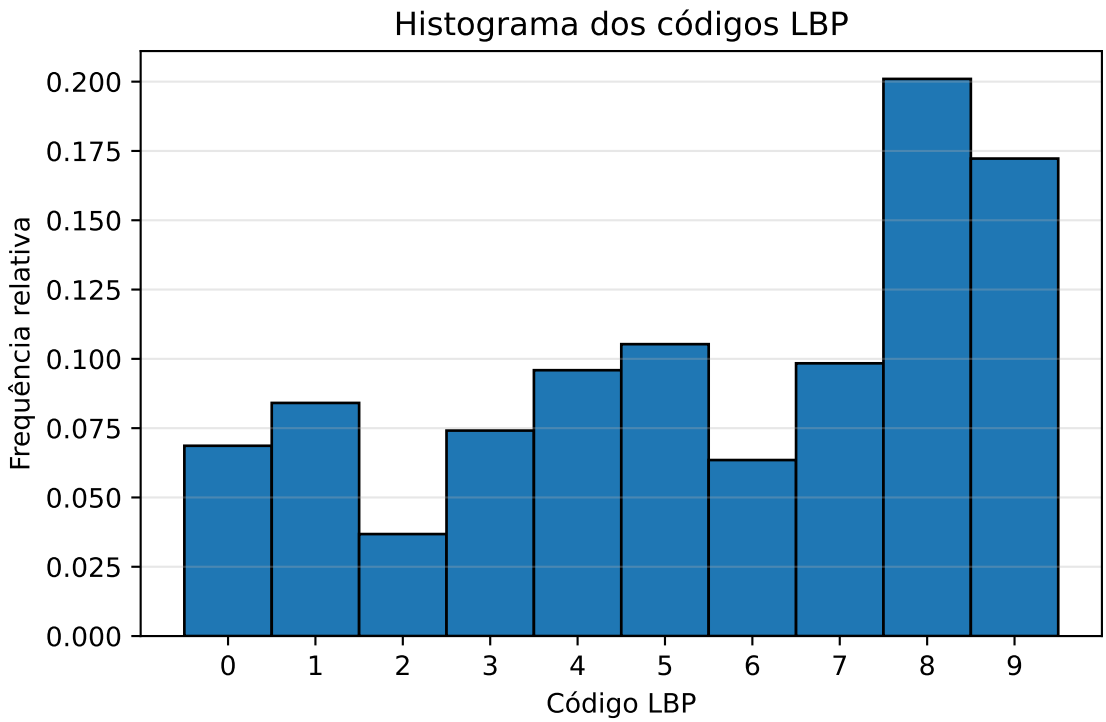


Figure 8.4: Histogramme des codes LBP de l'image du cameraman. Chaque barre représente la fréquence relative d'un code LBP, formant le vecteur de caractéristiques utilisé pour décrire la texture.

Fonction `local_binary_pattern`

L'implémentation utilisée dans ce chapitre est fournie par la bibliothèque `scikit-image` :

```
local_binary_pattern(
    imagem,
    P=8,
    R=1,
    method="uniform"
)
```

où :

- `image` : image en niveaux de gris ;
- `P` : nombre de voisins également espacés dans le voisinage circulaire ;
- `R` : rayon du voisinage, en pixels ;
- `method` : stratégie de codage. Dans ce chapitre, on utilise la valeur `"uniform"`.

L'équation présentée précédemment décrit le **LBP original**. Dans l'implémentation retenue dans ce chapitre, l'option `method="uniform"` calcule initialement ce code, puis remappe les motifs non uniformes vers une seule catégorie, réduisant la dimensionnalité du descripteur et le rendant plus robuste aux petites variations locales.

La Figure 8.3 présente la représentation visuelle du LBP, tandis que la Figure 8.4 montre l'histogramme des codes LBP utilisé comme vecteur de caractéristiques.

Le **Projet Pratique 2** (section **Comparaison de Descripteurs pour la Classification de Textures**) emploie le LBP dans la classification de différents types de texture synthétique.

8.6.2 *Histogram of Oriented Gradients* (HOG)

Le HOG est un descripteur qui représente la forme d'un objet à travers la distribution des orientations du gradient local. Comme pour l'opérateur de Canny (**Chapitre 6**), on calcule initialement le gradient :

$$|\nabla f(x,y)| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}, \qquad \theta(x,y) = \operatorname{atan2}\left(\frac{\partial f}{\partial y}, \frac{\partial f}{\partial x}\right).$$

où :

- $f(x,y)$ est l'intensité de l'image au pixel (x,y) ;
- $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial y}$ sont, respectivement, les dérivées partielles de l'image dans les directions horizontale et verticale ;
- $|\nabla f(x,y)|$ est la magnitude du vecteur gradient au pixel (x,y), indiquant l'intensité de la variation locale de l'image ;
- $\theta(x,y)$ est l'orientation du vecteur gradient au pixel (x,y), calculée par la fonction atan2, dont le résultat appartient à l'intervalle $(-\pi,\pi]$.

Bien que $\theta(x,y)$, tel que calculé par la fonction atan2, appartienne à l'intervalle $(-\pi,\pi]$, l'implémentation standard du HOG utilise le **gradient non signé** (*unsigned*) : les orientations opposées (par exemple, 0 et π) sont traitées comme équivalentes, et les angles sont mappés sur l'intervalle $[0,\pi)$ avant la construction de l'histogramme. Ce choix rend le descripteur invariant à la direction du contraste (par exemple, un bord clair-sombre et un bord sombre-clair produisent la même orientation).

L'image est ensuite divisée en **cellules** (*cells*). Pour chaque cellule, on construit un histogramme des orientations du gradient, pondéré par la magnitude correspondante. La concaténation des histogrammes de toutes les cellules forme le vecteur de caractéristiques HOG, qui représente la distribution spatiale des orientations du gradient et capture des informations sur la forme et les contours de l'objet (Figure 8.5).

```
n = 100

plt.figure(figsize=(8, 3))
plt.bar(
    range(n),
    hog_features[:n],
    width=0.9
)

plt.title("Primeiros componentes do vetor HOG")
plt.xlabel(f"Índice do componente (0-{n-1}, de um total de {hog_features.shape[0]})")
plt.ylabel("Valor normalizado")
plt.grid(axis="y", alpha=0.3)

plt.tight_layout()
plt.show()
```

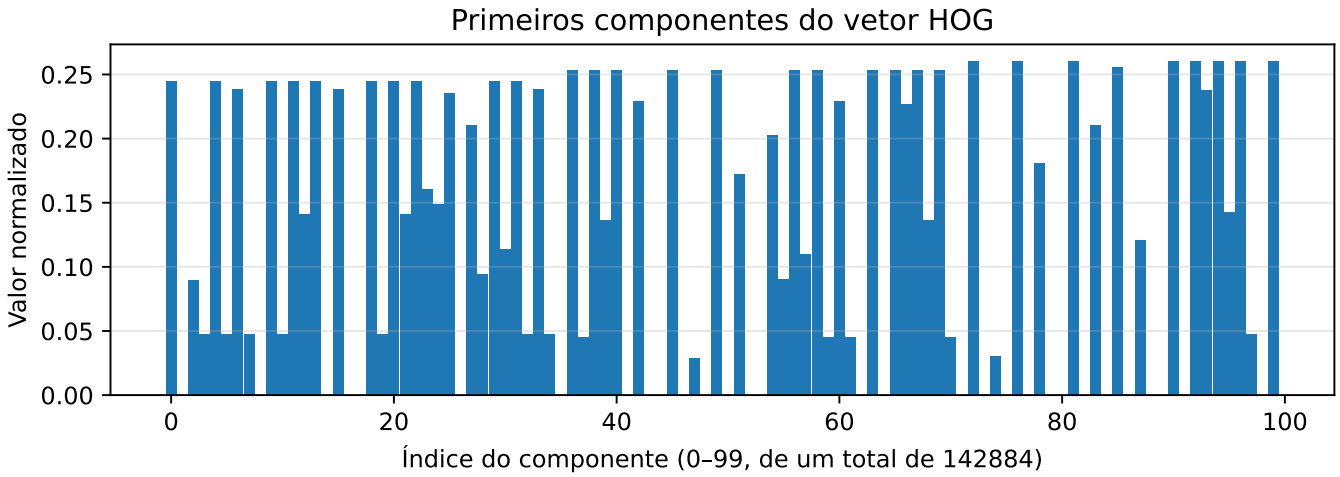


Figure 8.5: Premiers 100 composants du vecteur de caractéristiques HOG.

Fonction `hog`

L'extraction du descripteur HOG est réalisée par la fonction :

```
hog(
    image,
    orientations=9,
    pixels_per_cell=(8, 8),
    cells_per_block=(2, 2),
    visualize=True,
)
```

Les principaux paramètres sont :

- `image` : image d'entrée ;
- `orientations` : nombre de divisions angulaires de l'histogramme des orientations dans chaque cellule ;
- `pixels_per_cell` : taille, en pixels, de chaque cellule où l'histogramme est calculé ;
- `cells_per_block` : nombre de cellules utilisées pour la normalisation du descripteur ;
- `visualize` : lorsque `True`, retourne également une image illustrant les gradients utilisés par le HOG.

L'équation présentée précédemment décrit le calcul de la magnitude et de l'orientation du gradient, qui constituent la base du descripteur HOG. Dans l'implémentation adoptée dans ce chapitre, la fonction

`hog()` utilise ces informations pour construire des histogrammes d'orientations dans chaque cellule de l'image, puis effectue la normalisation par blocs (`cells_per_block`), réduisant la sensibilité du descripteur aux variations d'éclairage et de contraste.

La Figure 8.3 présente la représentation visuelle du HOG, tandis que la Figure 8.5 illustre les premiers composants du vecteur de caractéristiques extrait de l'image.

Le **Projet Pratique 1** (section **Classification de Chiffres Manuscrits avec k-NN**) compare les performances des descripteurs HOG avec l'utilisation directe des intensités des pixels comme vecteur de caractéristiques.

8.6.3 L'Impact de l'Échelle et la Normalisation des Caractéristiques

Le classificateur k-NN prend ses décisions en se basant sur la distance entre les vecteurs de caractéristiques. Par conséquent, l'échelle de chaque caractéristique influence directement le résultat de la classification. Si une variable présente des valeurs beaucoup plus grandes que les autres (par exemple, une intensité de couleur variant de 0 à 255, alors qu'un indice de circularité varie de 0 à 1), elle tend à dominer le calcul de la distance, réduisant ainsi l'influence des autres descripteurs.

Pour éviter ce problème, on applique une étape de **normalisation des caractéristiques**, généralement au moyen de la standardisation (*Z-score standardization*). Dans cette procédure, chaque caractéristique a désormais une moyenne égale à zéro et un écart-type égal à un, rendant comparables des grandeurs mesurées à l'origine sur des échelles différentes.

La standardisation est réalisée par la transformation

$$z = \frac{x - \mu}{\sigma},$$

où :

- x est la valeur originale de la caractéristique ;
- μ est la moyenne de cette caractéristique calculée sur l'ensemble d'entraînement ;
- σ est l'écart-type de la caractéristique ;
- z est la valeur standardisée.

Après cette transformation, toutes les caractéristiques possèdent une moyenne égale à zéro et un écart-type égal à un, ce qui leur permet de contribuer de manière équilibrée au calcul des distances.

Classe `StandardScaler`

La standardisation utilisée dans ce chapitre est réalisée par la classe `StandardScaler`, de la bibliothèque `scikit-learn` :

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_norm = scaler.fit_transform(X)
```

où :

- `StandardScaler()` : crée l'objet responsable de la standardisation ;
- `fit_transform(X)` : calcule la moyenne et l'écart-type de chaque caractéristique de l'ensemble `X` et retourne la matrice standardisée.

En pratique, la méthode `fit_transform()` exécute deux étapes : d'abord (`fit`), elle estime la moyenne (μ) et l'écart-type (σ) de chaque caractéristique ; ensuite (`transform`), elle applique la transformation de standardisation présentée précédemment à toutes les valeurs de la matrice d'entrée.

La Figure 8.6 montre l'effet de la normalisation. Visuellement, la distribution des points reste la même ; ce qui change, c'est l'échelle des axes. Sans normalisation, la caractéristique de plus grande magnitude domine le calcul des distances entre les échantillons. Après la standardisation, toutes les caractéristiques contribuent de manière équilibrée au calcul des distances utilisées par le classificateur k-NN.

```
# Dados sintéticos com escalas muito diferentes
np.random.seed(42)
X_demo = np.random.randn(20, 2) * [100, 1]
y_demo = np.array([0] * 10 + [1] * 10)

print("Effet de la normalisation :")
print("  Caractéristique 1 : échelle   100")
print("  Caractéristique 2 : échelle   1")
print("\nSans normalisation, la première caractéristique domine le calcul des distances.")
print("Avec normalisation, les deux contribuent de manière équilibrée.")
print("\nLa normalisation est essentielle lorsque les caractéristiques possèdent des échelles
différentes.")
fig, axes = plt.subplots(1, 2, figsize=(10, 4))

# Sem normalização
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()

# Com normalização
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()
```

```
Effet de la normalisation :
  Caractéristique 1 : échelle   100
  Caractéristique 2 : échelle   1

Sans normalisation, la première caractéristique domine le calcul des distances.
Avec normalisation, les deux contribuent de manière équilibrée.
```

```
La normalisation est essentielle lorsque les caractéristiques possèdent des échelles
différentes.
```

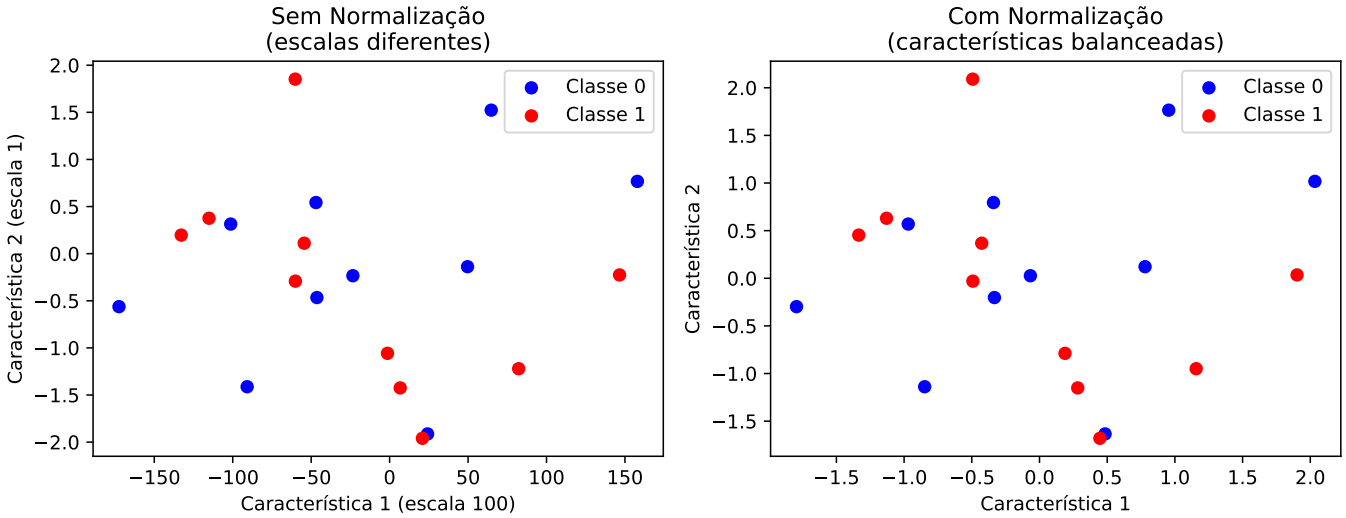


Figure 8.6: Importância da normalização das características para o classificador k-NN.

8.7 📌 Carte conceptuelle

Jusqu'ici, ont été présentés les descripteurs classiques (LBP, HOG, pixels bruts) et la manière dont ils organisent les échantillons dans un espace de caractéristiques. La Figure 8.7 synthétise ce parcours et situe ces étapes dans le flux général d'un système classique de classification d'images, en indiquant également les étapes suivantes — classification (k-NN) et évaluation des résultats — qui seront formalisées dans les sections suivantes.

8.8 Descripteurs en pratique

Après avoir pris connaissance des principaux descripteurs classiques, il est naturel de se demander comment ils influencent les performances d'un classificateur dans des situations proches de celles rencontrées en pratique.

Dans cette section, on compare l'utilisation de trois représentations distinctes des mêmes images : les intensités des pixels, les descripteurs LBP et les descripteurs HOG. Pour rendre l'expérience plus réaliste, on ajoute un bruit synthétique aux données, à des intensités différentes pour chaque descripteur — une forme simplifiée de simulation du fait qu'en pratique, différentes représentations tolèrent de manière inégale les imperfections de la capture (bruit du capteur, petites variations de position, etc.).

La Figure 8.8 présente les matrices de confusion obtenues pour chaque descripteur, permettant d'identifier dans quelles classes se produisent les principales erreurs de classification. L'interprétation de ces matrices a été introduite au **Chapitre 1**, lorsque les concepts de **Vrai Positif (VP)**, **Faux Positif (FP)**, **Vrai Négatif (VN)** et **Faux Négatif (FN)** ont été présentés. Ces concepts ont été explorés dans les **EPs 01_02** (métriques de classification) et **01_03** (*mean Average Precision* – mAP), disponibles sur :

- https://fzampirolli.github.io/pdi-vc/eps/py.pt/EP01_02.html
- https://fzampirolli.github.io/pdi-vc/eps/py.pt/EP01_03.html

Dans ce chapitre, les matrices de confusion sont employées pour analyser comment différents descripteurs influencent les performances du classificateur.

Ensuite, la Figure 8.9 résume la précision globale obtenue par chaque descripteur.

Les résultats montrent que les performances du classificateur dépendent directement de la représentation choisie pour décrire les images. Alors que l'utilisation directe des intensités des pixels est plus sensible aux

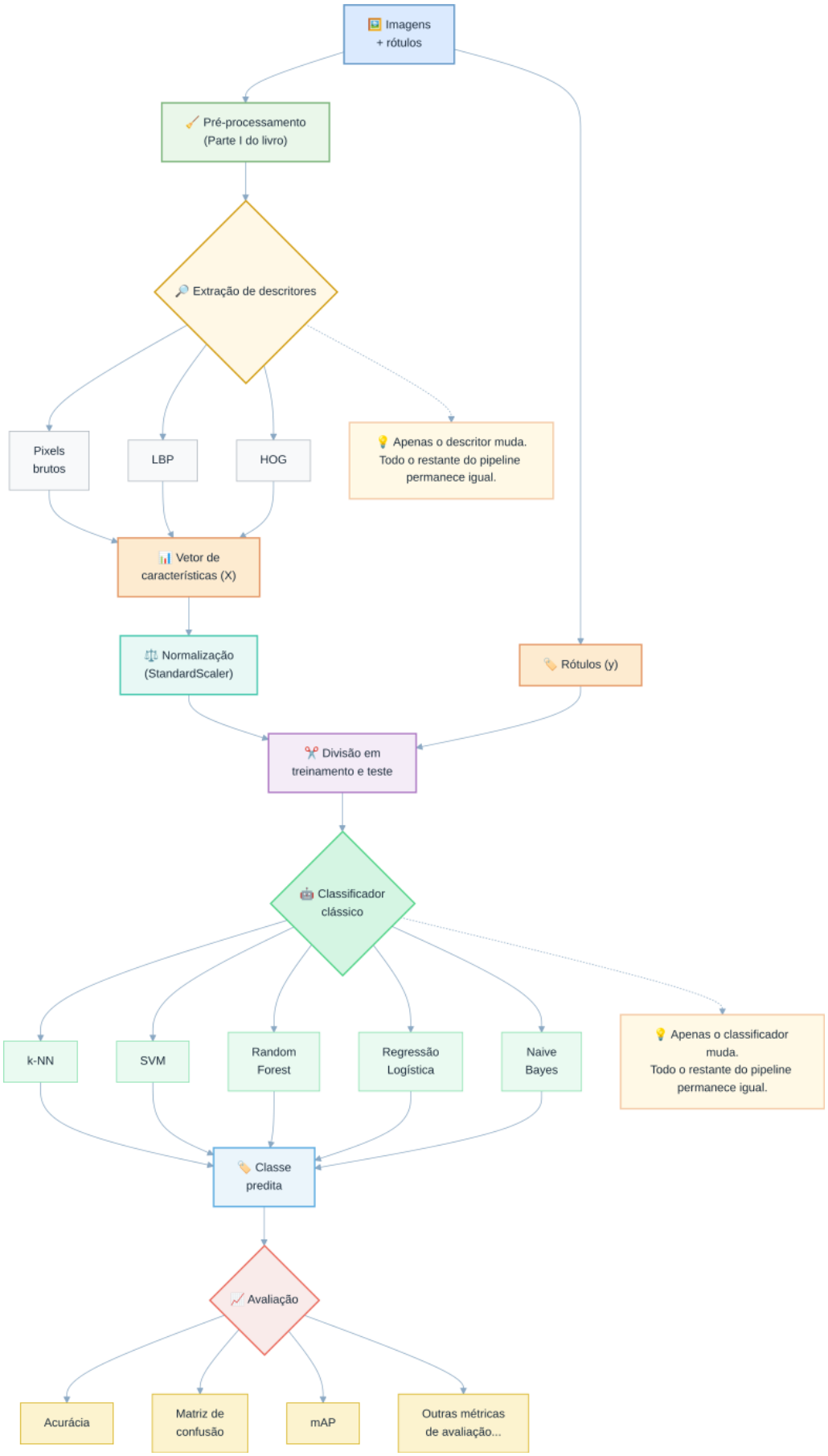


Figure 8.7: Carte conceptuelle du processus de classification d'images utilisant des descripteurs classiques (LBP, HOG, pixels bruts) et un classifieur traditionnel (k-NN). Ne s'applique pas aux modèles d'apprentissage profond (CNN, YOLO), qui apprennent des *features end-to-end* directement à partir des pixels.

dégradations introduites, les descripteurs LBP et HOG préservent mieux les informations pertinentes pour la classification, ce qui se traduit par de meilleures performances dans ce scénario. Il est important de souligner que les niveaux de bruit appliqués à chaque descripteur ont été choisis uniquement à des fins pédagogiques, afin d'illustrer le principe général selon lequel des descripteurs plus élaborés *peuvent* être plus robustes aux dégradations — ce qui ne signifie pas que cette relation se vérifie toujours, comme l'étude de cas de la section suivante le démontrera.

Comment l'expérience est réalisée

Comme l'objectif de cette section est de comparer uniquement l'effet des descripteurs, on génère un ensemble de données synthétiques simple : trois nuages de points gaussiens, centrés sur les mêmes valeurs de « couleur, texture et forme » déjà utilisées dans la Figure 8.1 — le même motif employé depuis le début du chapitre pour représenter les trois classes de fruits.

```
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)
```

où :

- `centros` : dictionnaire contenant le point moyen de chaque classe dans l'espace des caractéristiques (couleur, texture, forme) ;
- `n_por_classe` : nombre d'échantillons générés par classe ;
- `np.random.normal(centro, 0.12, (n_por_classe, 3))` : génère `n_por_classe` échantillons autour de chaque centre, avec un écart-type de 0,12 dans chaque dimension ;
- `np.repeat(list(centros.keys()), n_por_classe)` : génère le vecteur d'étiquettes correspondant, dans le même ordre que les centres.

Ensuite, on utilise le flux d'entraînement et d'évaluation :

- `train_test_split(X, y, test_size=0.3)` : divise les données en entraînement (70 %) et test (30 %) ;
- `KNeighborsClassifier(n_neighbors=5)` : crée un classificateur k-NN avec $k = 5$ voisins ;
- `fit(X_train, y_train)` : ajuste le modèle aux données d'entraînement ;
- `predict(X_test)` : classe les échantillons de test ;
- `accuracy_score(y_test, y_pred)` : calcule la précision ;
- `confusion_matrix(y_test, y_pred)` : génère la matrice de confusion.

Dans cette expérience, l'ensemble d'entraînement, le classificateur et la méthode d'évaluation restent exactement les mêmes. La seule différence entre les expériences réside dans la représentation utilisée pour chaque image (pixels bruts, LBP ou HOG), permettant d'évaluer exclusivement l'influence du descripteur sur la performance du classificateur.

```
classes = ["Maçã", "Banana", "Laranja"]

np.random.seed(42)
```

```
# Mêmes centres de classes (couleur, texture, forme) utilisés dans l'
# exemple précédent, maintenant réutilisés pour générer les données
# synthétiques d'entraînement et de test de cette expérience.
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)

# Simuler des descripteurs avec différents niveaux de sensibilité au bruit.
# Plus le bruit ajouté est élevé, plus la représentation tend à être mauvaise.
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),
}

# Créer une seule figure avec 3 sous-graphiques côte à côte pour les 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

    # Matrice de confusion dans le sous-graphique correspondant

    cm = confusion_matrix(y_test, y_pred, labels=classes)

    sns.heatmap(
        cm,
        annot=True,
        fmt="d",
        cmap="Blues",
        cbar=False,
        xticklabels=classes,
        yticklabels=classes,
        ax=axes[idx],
    )

    axes[idx].set_title(
        f"{nome}\nAcurácia: {acc:.3f}",
        fontsize=11,
        fontweight="bold"
    )
    axes[idx].set_xlabel("Classe Predita")
    axes[idx].set_ylabel("Classe Real")
```

```
plt.tight_layout()
plt.show()
```

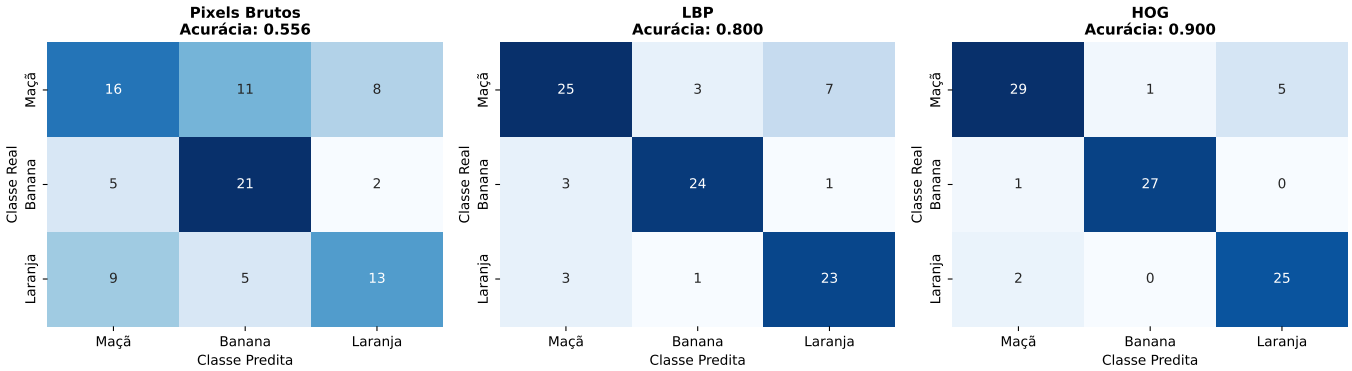


Figure 8.8: Matrices de confusion obtenues par le classifieur k-NN utilisant trois descripteurs différents. Les lignes représentent la classe réelle (Pomme, Banane et Orange) et les colonnes la classe prédite. Plus la concentration de valeurs sur la diagonale principale est élevée, meilleure est la performance du descripteur.

```
# Comparaison visuelle sur une figure isolée
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)

# Ajouter les valeurs au-dessus des barres en utilisant round standard pour l'affichage
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("Analyse des résultats :")
print("- Pixels bruts : sensibles aux variations locales d'éclairage et au bruit.")
print("- LBP : bonne tolérance aux variations monotones de l'éclairage global.")
print("- HOG : excellent pour les contours et les formes stables sous de petites fluctuations
géométriques.")
```

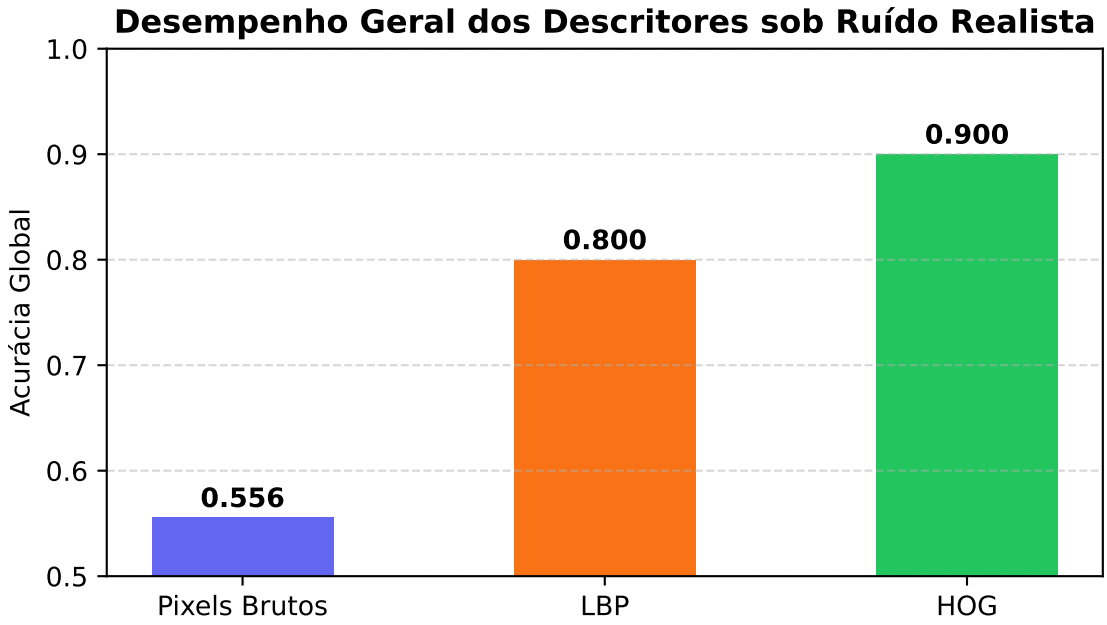


Figure 8.9: Comparaison détaillée de l'exactitude globale dans un scénario réaliste. Observez comment les descripteurs extraits structurellement (LBP et HOG) surpassent l'utilisation des intensités pures des pixels bruts.

```
Analyse des résultats :
- Pixels bruts : sensibles aux variations locales d'éclairage et au bruit.
- LBP : bonne tolérance aux variations monotones de l'éclairage global.
- HOG : excellent pour les contours et les formes stables sous de petites fluctuations
géométriques.
```

8.9 Um Problema Concreto : Simulation de Descripteurs de Fruits

Reprenant le problème de classification des fruits présenté au début du chapitre, chaque image peut être représentée par un vecteur de caractéristiques (*feature vector*) obtenu à partir de l'extraction de descripteurs de couleur, de texture et de forme. La Table 8.1 présente quelques descripteurs fréquemment utilisés dans les applications de Vision par Ordinateur, y compris des techniques introduites dans le Chapitre 3 et dans ce chapitre.

Tableau 8.1: Ensemble de descripteurs chromatiques, texturaux et géométriques utilisés pour représenter des images de fruits.

Caractéristique	Description
R, G, B	intensité moyenne des canaux rouge, vert et bleu
NC	intensité moyenne en niveaux de gris (*grayscale*)
LBP	descripteur de texture (*Local Binary Pattern*)
HOG	descripteur de forme (*Histogram of Oriented Gradients*)
Aire	nombre de pixels de l'objet
Périmètre	longueur du contour
Circularité	mesure de la circularité de l'objet
Rapport largeur/hauteur	proportion entre la largeur et la hauteur de la région

Dans cet exemple, chaque image est représentée par le vecteur

$$X = (R, G, B, NC, \text{LBP}, \text{HOG}, \text{Área}, \text{Perímetro}, \text{Circularidade}, \text{Razão}).$$

Simplification adoptée dans ce tableau

En pratique, LBP et HOG ne sont pas des valeurs scalaires, mais des histogrammes comportant des dizaines ou des centaines de composantes. Dans cette section, chacun d'eux est représenté par une valeur unique uniquement pour simplifier la présentation. Dans les applications réelles, ces positions seraient remplacées par les composantes complètes des histogrammes respectifs.

Dans les applications réelles, tous les descripteurs ne contribuent pas de manière égale à distinguer les classes. Certains fournissent des informations plus pertinentes, tandis que d'autres peuvent être redondants ou peu discriminatifs.

Pour reproduire ce scénario de manière contrôlée, on utilisera `make_classification()`, de la bibliothèque `scikit-learn`. La fonction génère un ensemble de données synthétiques dont les caractéristiques peuvent être interprétées comme des descripteurs d'images, permettant de définir combien d'entre elles seront informatives pour la classification.

Dans cet exemple, dix caractéristiques synthétiques sont générées, dont seulement sept (`n_informative=7`) participent à la séparation entre les trois classes. Les autres simulent des attributs peu informatifs ou redondants. La Figure 8.10 présente une représentation conceptuelle de ce processus.

Avant d'introduire l'algorithme qui sera étudié en détail dans ce chapitre, il convient de faire une observation : pour identifier, parmi les dix caractéristiques synthétiques, lesquelles sont les plus discriminatives — et ainsi sélectionner deux d'entre elles pour la visualisation en 2D —, on utilise un *Random Forest* uniquement comme outil auxiliaire de diagnostic. Le KNN, objet de ce chapitre, est présenté ci-après.

```
# Configuration pour la reproduction
np.random.seed(42)

# Génération de données synthétiques avec des caractéristiques contrôlées
X, y = make_classification(
    n_samples=300,
    n_features=10,
    n_informative=7,
    n_redundant=2,
    n_repeated=1,        # Une caractéristique est une copie d'une autre
    n_classes=3,
    n_clusters_per_class=1,
    random_state=42,
)

# Créer une figure avec deux sous-graphiques
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# Sous-graphique 1 : Visualisation des classes en 2D (en utilisant deux caractéristiques
informatives)
# Identifier quelles caractéristiques sont les plus informatives
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X, y)
importancias = rf.feature_importances_
caracteristicas_informativas = np.argsort(importancias)[-2:]  # Les deux plus importantes

cores = {0: "#e74c3c", 1: "#f1c40f", 2: "#e67e22"}  # Pomme, Banane, Orange
rotulos = {0: "Maçã", 1: "Banana", 2: "Laranja"}

for classe in range(3):
    idx = y == classe
```

```
    ax1.scatter(X[idx, caracteristicas_informativas[0]],
                X[idx, caracteristicas_informativas[1]],
                c=cores[classe], label=rotulos[classe],
                alpha=0.6, s=50, edgecolors="white", linewidth=0.5)

ax1.set_xlabel(f"Característica {caracteristicas_informativas[0]+1} (informativa)",
fontsize=11)
ax1.set_ylabel(f"Característica {caracteristicas_informativas[1]+1} (informativa)",
fontsize=11)
ax1.set_title("Classes no Espaço de Características\n(2 características informativas)",
             fontsize=12, fontweight="bold")
ax1.legend(loc="upper right")
ax1.grid(alpha=0.3)

# Sous-graphique 2 : Importance des caractéristiques
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)

# Colorer les barres pour mettre en évidence les caractéristiques
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)

# Ajouter une légende
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")

# Annoter le nombre de caractéristiques informatives
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  Analyse des données générées :")
print(f"  • Total d'échantillons : {X.shape[0]}")
print(f"  • Nombre de caractéristiques : {X.shape[1]}")
print(f"  • Caractéristiques informatives : 7 (colonnes 1 à 7 du graphique)")
print(f"  • Caractéristiques redondantes : 2 (colonnes 8 et 9)")
print(f"  • Caractéristiques répétées : 1 (colonne 10)")
print(f"  • Distribution des classes : {np.bincount(y)}")
```

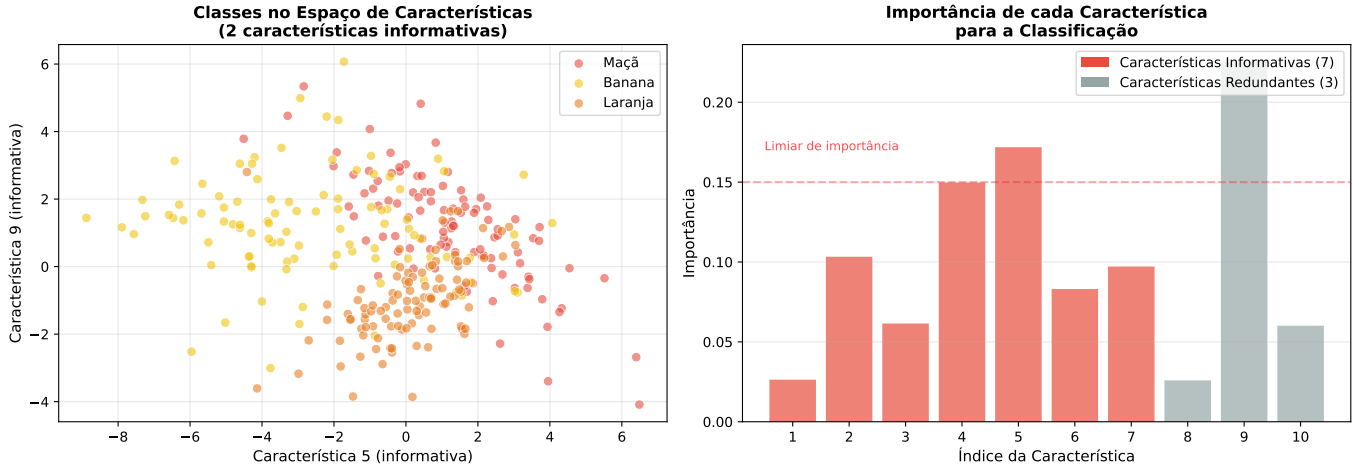


Figure 8.10: Illustration du processus de génération de données synthétiques avec *make_classification*. À gauche, visualisation des trois classes dans un espace bidimensionnel formé par deux caractéristiques informatives. À droite, importance relative de chaque caractéristique pour la classification, montrant que seulement 7 des 10 caractéristiques sont effectivement discriminantes, tandis que les autres sont redondantes (2) ou répétées (1).

```
Analyse des données générées :
• Total d'échantillons : 300
• Nombre de caractéristiques : 10
• Caractéristiques informatives : 7 (colonnes 1 à 7 du graphique)
• Caractéristiques redondantes : 2 (colonnes 8 et 9)
• Caractéristiques répétées : 1 (colonne 10)
• Distribution des classes : [101  98 101]
```

8.10 Classificateur k-NN : Comment il fonctionne en interne

Le **k-*Nearest Neighbors* (k-NN)** est l'un des algorithmes de classification les plus simples et intuitifs de l'apprentissage automatique. Contrairement à de nombreux classificateurs, il ne construit pas explicitement un modèle pendant la phase d'entraînement. Au lieu de cela, il stocke les échantillons étiquetés et, lorsqu'un nouvel échantillon doit être classifié, il recherche ceux qui lui ressemblent le plus.

Le principe de l'algorithme repose sur l'hypothèse que des échantillons présentant des caractéristiques similaires ont tendance à appartenir à la même classe. Pour quantifier cette proximité, le k-NN utilise une mesure de distance entre les vecteurs de caractéristiques.

À titre d'exemple, considérons la Table 8.2, qui présente une version simplifiée du problème de classification des fruits utilisant seulement deux caractéristiques : l'intensité de la couleur et la circularité, toutes deux normalisées dans l'intervalle de 0 à 1.

Tableau 8.2: Exemple simplifié de classification de fruits utilisant deux caractéristiques normalisées.

Échantillon	Couleur	Circularité	Classe
Fruit 1	0,82	0,88	Pomme
Fruit 2	0,30	0,20	Banane
Fruit 3	0,88	0,85	Pomme
Fruit ? (test)	0,80	0,90	?

En observant uniquement ces deux caractéristiques, on remarque que le fruit de test est beaucoup plus proche des échantillons étiquetés comme **Pomme** que de l'échantillon étiqueté comme **Banane**. Dans la section

suivante, cette notion intuitive de proximité sera formalisée au moyen d'une métrique de distance, utilisée par l'algorithme pour identifier les voisins les plus proches et décider de la classe du nouvel échantillon.

8.10.1 Métrique de distance

La proximité entre deux échantillons est généralement quantifiée par la **distance euclidienne**, définie par

$$d(x, x_i) = \|x - x_i\|_2 = \sqrt{\sum_{j=1}^{n} (x_j - x_{i,j})^2},$$

où :

- x est l'échantillon de test ;
- x_i est un échantillon de l'ensemble d'entraînement ;
- n est le nombre de caractéristiques ;
- x_j et $x_{i,j}$ représentent la j-ième caractéristique.

Dans l'implémentation de ce chapitre, x correspond à une ligne de `X_test` et x_i à une ligne de `X_train`. La méthode `predict()` calcule automatiquement la distance entre x et tous les échantillons d'entraînement.

Dans l'exemple de la Table 8.2:

$$d(\text{teste}, \text{Fruta 1}) \approx 0{,}028, \qquad d(\text{teste}, \text{Fruta 2}) \approx 0{,}860, \qquad d(\text{teste}, \text{Fruta 3}) \approx 0{,}094.$$

Comme les plus petites distances correspondent aux Fruits 1 et 3, ces échantillons seront utilisés lors de l'étape de décision.

8.10.2 Règle de décision

Après avoir trié les distances, l'algorithme sélectionne les k voisins les plus proches. Soit $N_k(x)$ cet ensemble. La classe prédite est donnée par

$$\hat{y} = \text{moda}\{\, y_i : x_i \in N_k(x) \,\},$$

où y_i est le label de l'échantillon x_i et $\hat{y}$ est la classe attribuée à l'échantillon de test.

Dans l'exemple, pour $k = 3$, les voisins sont Fruit 1 (Pomme), Fruit 3 (Pomme) et Fruit 2 (Banane). Comme **Pomme** reçoit deux votes, c'est la classe prédite.

Classe `KNeighborsClassifier`

Dans ce chapitre, l'algorithme est implémenté avec la classe `KNeighborsClassifier` de la bibliothèque `scikit-learn` :

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

y_pred = knn.predict(X_test)
```

où :

- `KNeighborsClassifier(n_neighbors=3)` : définit la valeur de k ;
- `fit(X_train, y_train)` : stocke les échantillons d'entraînement (`X_train`) et leurs étiquettes (`y_train`) ;
- `predict(X_test)` : retourne les classes prédites pour les échantillons de `X_test`.

En interne, `predict()` exécute les étapes décrites précédemment : calcule les distances, identifie les k voisins les plus proches et détermine la classe par vote majoritaire.

La Figure 8.11 illustre cette procédure sur un ensemble bidimensionnel. La figure met en évidence les voisins utilisés lors de la classification, tandis que la console présente les étapes de l'algorithme : calcul des distances, tri, sélection des voisins, vote et prédiction de la classe.

```
def knn_passo_a_passo(X, y, x, k=3):
    """Exécute les cinq étapes de l'algorithme k-NN.
    Paramètres : X (entraînement), y (étiquettes), x (test) et k (nombre de voisins).
    """
    # 1. Distances
    dist = [(np.linalg.norm(x-xi), yi, i) for i, (xi, yi) in enumerate(zip(X, y))]
    print(f"1. Distances calculées : {len(dist)}")

    # 2. Tri
    dist.sort(key=lambda t: t[0])
    print("2. Distances triées")

    # 3. Sélection
    vizinhos = dist[:k]
    print(f"3. {k} voisins les plus proches :")
    for d, c, _ in vizinhos:
        print(f"   {d:.4f} → {c}")

    # 4. Vote
    votos = {}
    for _, c, _ in vizinhos:
        votos[c] = votos.get(c, 0) + 1
    print("4. Votes :", votos)

    # 5. Décision
    classe = max(votos, key=votos.get)
    print("5. Classe prédite :", classe)

    return classe, vizinhos


# Données d'exemple
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)

# Visualisation
plt.figure(figsize=(5,5))

for c, rotulo in [("Classe A","Classe A"), ("Classe B","Classe B")]:
    P = X[y==c]
    plt.scatter(P[:,0], P[:,1], s=80, label=rotulo)

plt.scatter(*x, marker="*", s=220, edgecolors="black", label="Teste")

for _, _, i in vizinhos:
    plt.scatter(*X[i], s=220, facecolors="none", edgecolors="black", linewidths=2)
    plt.plot([x[0], X[i,0]], [x[1], X[i,1]], "--", lw=1)
```

```
plt.xlabel("Característica 1")
plt.ylabel("Característica 2")
plt.title(f"k-NN ($k=3$): classe predita = {classe}")
plt.legend()
plt.grid(alpha=.3)
plt.axis("equal")
plt.tight_layout()
plt.show()
```

```
1. Distances calculées : 30
2. Distances triées
3. 3 voisins les plus proches :
   1.0850 → Classe A
   1.6379 → Classe B
   1.6382 → Classe A
4. Votes : {np.str_('Classe A'): 2, np.str_('Classe B'): 1}
5. Classe prédite : Classe A
```

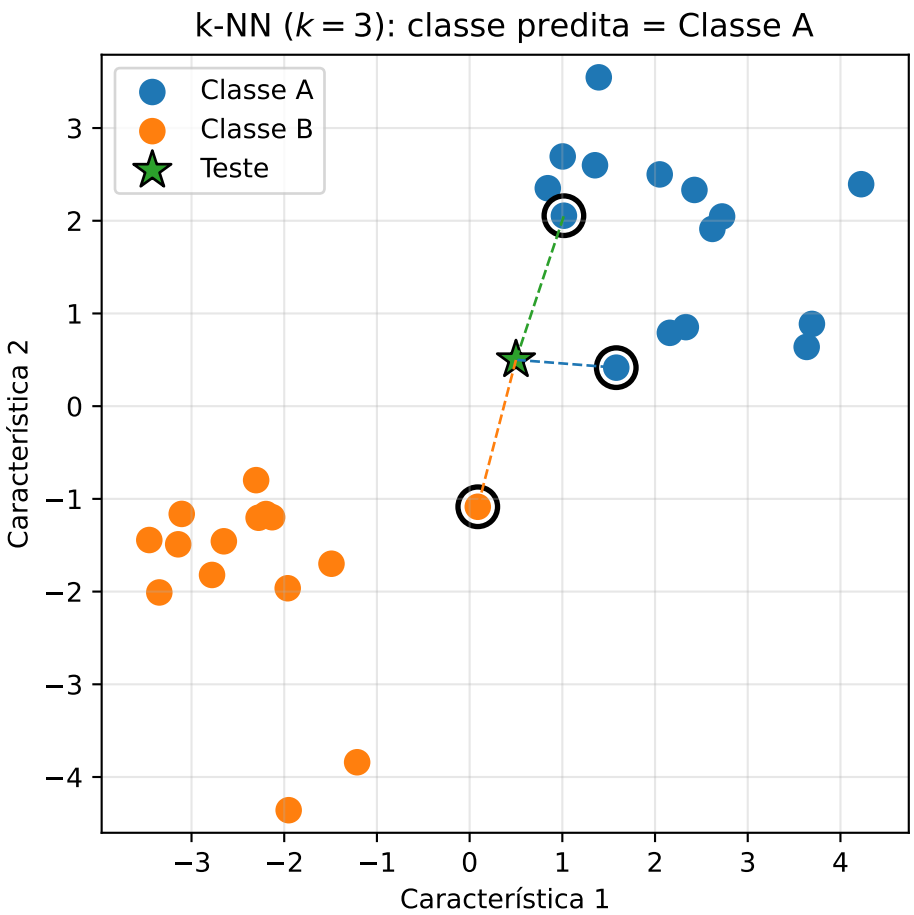


Figure 8.11: Classification d'un nouvel échantillon par l'algorithme k-NN. Le point de test (étoile) est classé à partir des trois voisins les plus proches, mis en évidence par des cercles.

8.10.3 Le rôle du paramètre k

Le paramètre k détermine combien de voisins participent à la décision de classification.

- **Les petites valeurs de** k (par exemple, $k = 1$) rendent le classificateur plus sensible aux bruits et aux variations locales, produisant des frontières de décision plus irrégulières et favorisant le *surapprentissage* (*overfitting*).

- **Les grandes valeurs de** k produisent des frontières de décision plus lisses, mais peuvent réduire la sensibilité aux structures locales, favorisant le *sous-apprentissage* (*underfitting*).

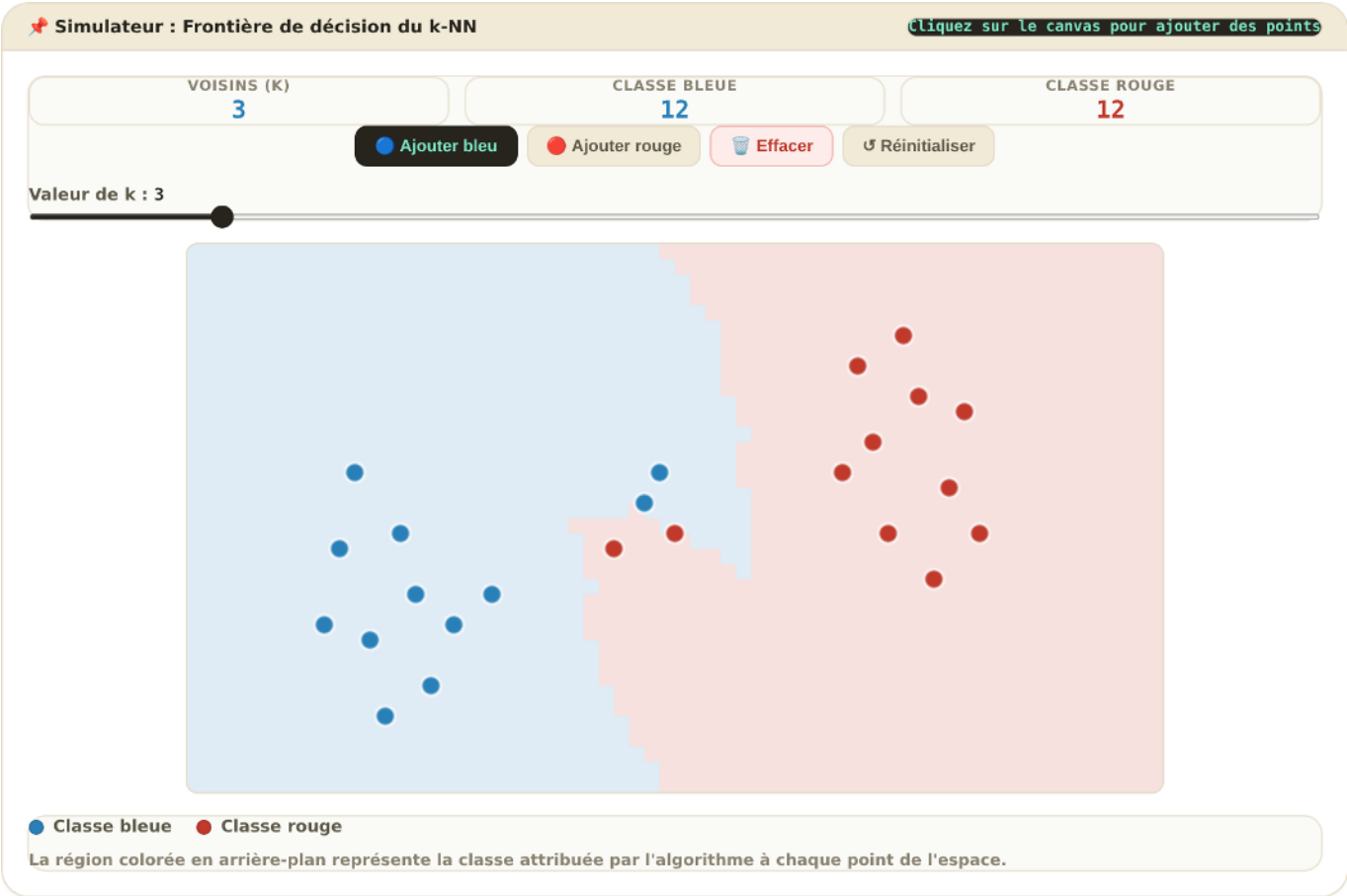
Dans les problèmes à deux classes, il est courant d'utiliser des valeurs impaires de k afin de réduire l'occurrence d'égalités.

Un autre aspect important est la **malédiction de la dimensionnalité** (*curse of dimensionality*). À mesure que le nombre de caractéristiques augmente, les distances entre les échantillons tendent à devenir plus similaires, rendant difficile l'identification de voisins réellement représentatifs.

Résumé

L'algorithme k-NN peut être résumé en trois étapes :
1. extraire le vecteur de caractéristiques du nouvel échantillon ;
2. identifier les k voisins les plus proches ;
3. classer l'échantillon selon la classe la plus fréquente parmi ces voisins.

Le simulateur de la Figure 8.12 permet d'explorer visuellement l'effet du paramètre k sur la frontière de décision.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap07/sim-07-knn.html

Figure 8.12: Simulateur interactif de la frontière de décision du k-NN : ajoutez des points d'entraînement et ajustez la valeur de k pour observer l'effet sur la région de décision.

8.11 Projet Pratique 1 : Classification de Chiffres Manuscrits avec k-NN

Les sections précédentes ont présenté l'algorithme k-NN à travers un exemple simplifié de classification de fruits, n'utilisant que deux caractéristiques. Ci-après, le même algorithme est appliqué à un ensemble de données d'images, dans lequel chaque échantillon est représenté par un vecteur de plus grande dimension.

Comme étude de cas, on utilise la base publique `load_digits`, mise à disposition par la bibliothèque `scikit-learn`. Cet ensemble de données contient 1797 images de chiffres manuscrits des classes de 0 à 9, chacune ayant une résolution de 8×8 pixels en niveaux de gris. Chaque image est représentée par un vecteur de 64 caractéristiques, correspondant aux intensités des pixels, et chaque vecteur possède une étiquette indiquant le chiffre correspondant.

La base `load_digits` est mise à disposition par la bibliothèque `scikit-learn` et est utilisée dans ce chapitre pour illustrer l'application de l'algorithme k-NN. En plus d'être disponible directement dans `scikit-learn`, elle dispense d'étapes supplémentaires d'obtention et de préparation des données, ce qui permet de concentrer l'attention sur l'implémentation et l'évaluation du classifieur.

La Figure 8.13 présente un échantillon des images de la base de données.

```
digits = load_digits()
print(f"Total d'échantillons : {digits.data.shape[0]}, dimension du vecteur :
{digits.data.shape[1]}")
print(f"Classes : {[int(i) for i in sorted(set(digits.target))]}")

n_amostras = 16
imgs = list(digits.images[:n_amostras])
imgs_titles = [str(label) for label in digits.target[:n_amostras]]
mm.show(imgs, titles=imgs_titles, cols=8, figsize=(12, 4))
```

```
Total d'échantillons : 1797, dimension du vecteur : 64
Classes : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

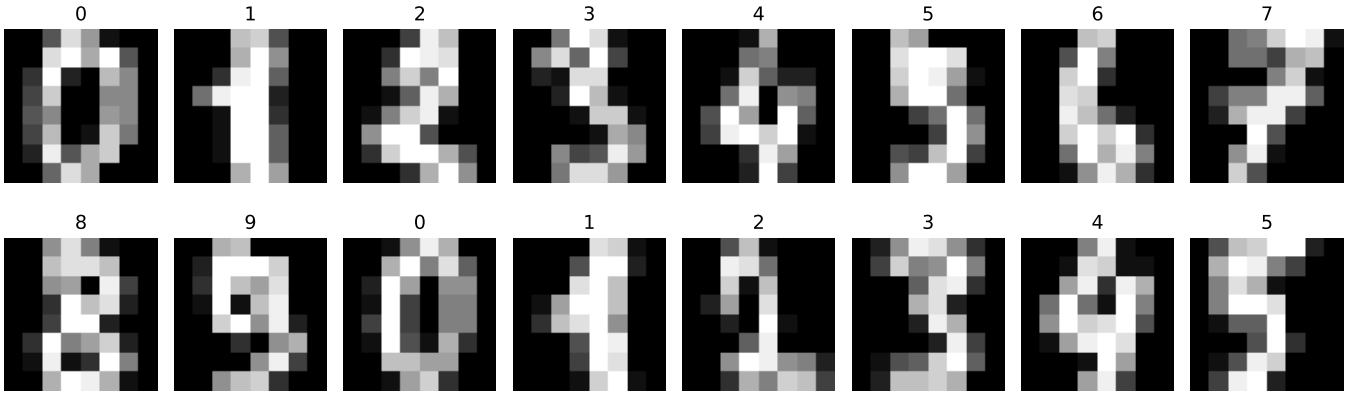


Figure 8.13: Échantillon de chiffres manuscrits de la base *load_digits*, utilisé comme étude de cas de classification.

8.11.1 Classification avec des vecteurs d'intensité

Dans cette première expérience, chaque image de dimension 8×8 est représentée directement par les intensités de ses 64 pixels, sans extraction de descripteurs supplémentaires. Ainsi, chaque échantillon correspond à un vecteur de 64 caractéristiques, utilisé comme entrée du classifieur k-NN.

Ensuite, l'ensemble de données est divisé en sous-ensembles d'entraînement et de test, en préservant la proportion des dix classes au moyen du paramètre `stratify=y`. Le classifieur est entraîné avec $k = 3$ et évalué sur l'ensemble de test en utilisant la précision et la matrice de confusion présentée dans la Figure 8.14..

```
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"Précision (vecteurs d'intensité, 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()
```

```
Précision (vecteurs d'intensité, k=3): 0.9870
```

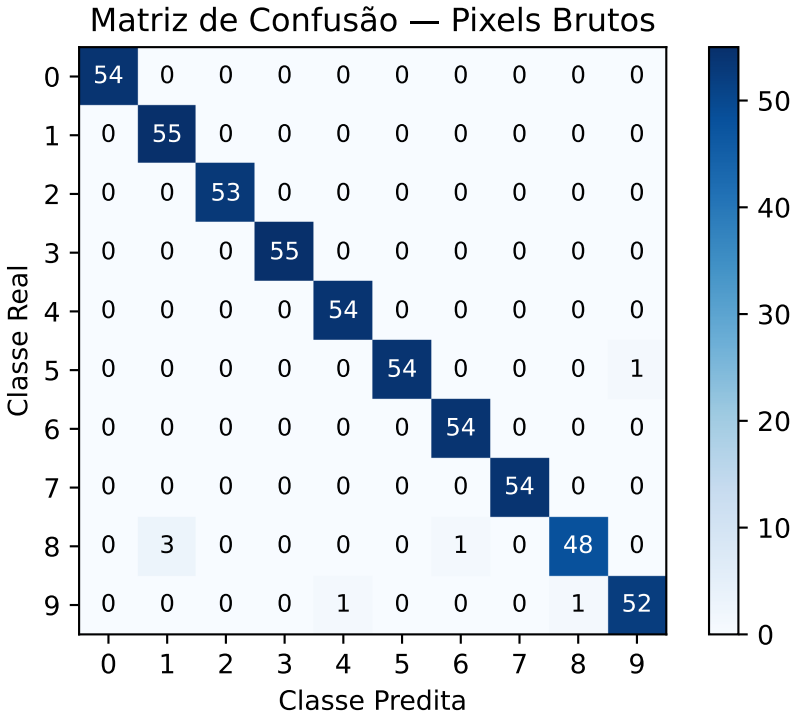


Figure 8.14: Matrice de confusion du classificateur k-NN entraîné avec des vecteurs d'intensité bruts (pixels).

8.11.2 Classification avec des descripteurs HOG

Dans l'expérience précédente, chaque image était représentée directement par les intensités de ses pixels. Dans cette section, cette représentation est remplacée par des descripteurs HOG (*Histogram of Oriented Gradients*), qui encodent des informations sur la distribution des orientations des gradients de l'image.

On conserve la même partition des données, le même classificateur k-NN et le même protocole d'évaluation, en modifiant uniquement la représentation des images. La Figure 8.15 compare les résultats obtenus avec des vecteurs d'intensité et avec des descripteurs 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"Dimension du vecteur 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"Précision (descripteur 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)  # Augmente la limite supérieure pour laisser de l'espace
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")  # Augmente le décalage vertical
plt.tight_layout()
```

```
Dimension du vecteur HOG : 32
Précision (descripteur HOG, k=3) : 0.7593
```

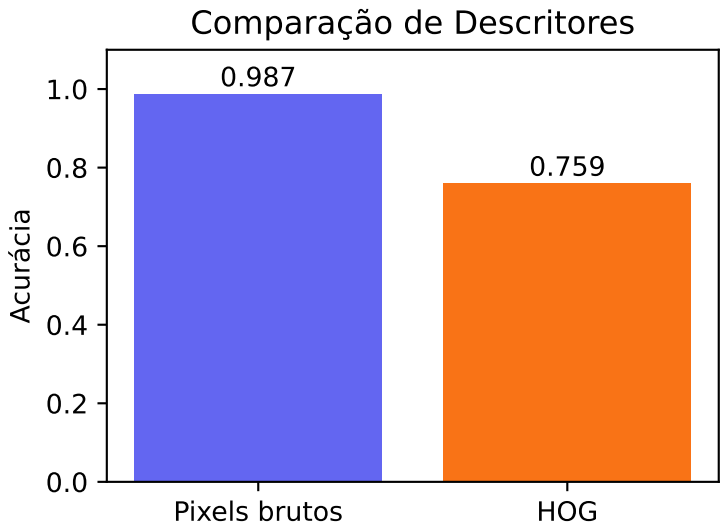


Figure 8.15: Comparaison de précision entre les descripteurs de pixels bruts et HOG pour le classificateur k-NN (k=3) sur la base de chiffres.

Pourquoi cela se produit-il ?

Dans la Figure 8.15, le classifieur entraîné avec des vecteurs d'**intensité des pixels** atteint une précision plus élevée (0.987) que celui basé sur des descripteurs **HOG** (0.759). Ce résultat est lié aux caractéristiques de la base `load_digits`.
Les images ont une résolution de seulement 8×8 pixels, sont approximativement centrées et présentent peu de variation d'éclairage, d'échelle et d'orientation. Dans ce scénario, les intensités des pixels préservent pratiquement toute l'information nécessaire pour distinguer les classes. En revanche, le HOG résume l'image en histogrammes d'orientations des gradients, réduisant une partie du détail spatial disponible dans les pixels d'origine.
Cette réduction d'information peut rendre difficile la séparation de chiffres visuellement similaires, comme 3 et 8 ou 4 et 9, surtout lorsque la résolution de l'image est faible.
Dans des problèmes avec des images de plus haute résolution ou sujettes à des variations d'éclairage, de position, d'échelle ou de petites déformations, des descripteurs comme le HOG tendent à mieux représenter la structure locale de l'image que les valeurs individuelles des pixels. Ainsi, cette expérience illustre un principe important de l'apprentissage automatique : **la représentation des données doit être choisie en fonction des caractéristiques du problème et non de la complexité du descripteur.**

8.12 Évaluation des Classifieurs

Dans les sections précédentes, la qualité du classifieur a été analysée à l'aide de l'exactitude et de la matrice de confusion. Dans cette section, ces outils sont complétés par des métriques utilisées pour l'évaluation des modèles et par une procédure permettant de sélectionner la valeur du paramètre k.

L'exactitude correspond à la proportion d'échantillons correctement classifiés. Bien qu'elle soit une mesure simple et largement utilisée, elle peut s'avérer insuffisante lorsque les classes présentent des distributions très déséquilibrées.

À partir de la matrice de confusion — introduite dans le **Chapitre 1** et utilisée tout au long de ce chapitre — des métriques par classe peuvent être calculées, comme la précision et le rappel :

$$\text{Précision} = \frac{VP}{VP + FP}, \qquad \text{Rappel} = \frac{VP}{VP + FN},$$

où VP, FP et FN représentent, respectivement, le nombre de vrais positifs, de faux positifs et de faux négatifs de la classe analysée. La précision quantifie la proportion de prédictions positives correctes, tandis que le rappel mesure la capacité du classifieur à identifier les exemples appartenant à la classe.

8.12.1 Choix de k par Validation Croisée

Dans les expériences précédentes, on a adopté $k = 3$ pour illustrer le fonctionnement de l'algorithme. Cependant, ce paramètre influence directement la performance du classificateur et, en pratique, doit être sélectionné à partir des données.

Une approche largement utilisée est la **validation croisée** (*cross-validation*), dans laquelle l'ensemble d'entraînement est divisé en partitions successives pour estimer la performance du modèle sur des données non utilisées pendant l'entraînement.

Le code suivant calcule la précision moyenne obtenue par validation croisée de cinq partitions (*5-fold cross-validation*) pour différentes valeurs de k. La Figure 8.16 présente les résultats, permettant d'identifier la région où le classificateur atteint la meilleure performance.

```
valores_k = range(1, 16)
acuracias_medias = []
```

```
for k in valores_k:
    modelo = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(modelo, X, y, cv=5)
    acuracias_medias.append(scores.mean())

melhor_k = list(valores_k)[int(np.argmax(acuracias_medias))]
print(f"Meilleure valeur de k trouvée : {melhor_k} (précision
moyenne={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()
```

```
Meilleure valeur de k trouvée : 2 (précision moyenne=0.9672)
```

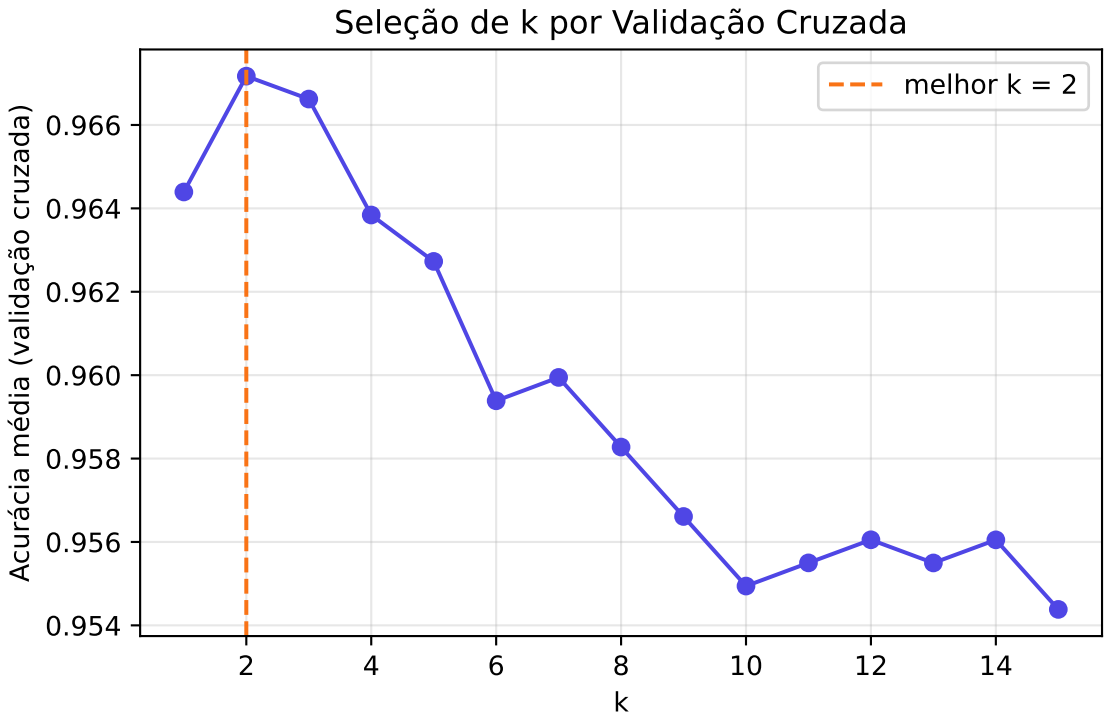


Figure 8.16: Précision moyenne par validation croisée (5 partitions) en fonction du paramètre k, pour la base de chiffres avec vecteurs d'intensité bruts.

8.12.2 Le compromis entre biais et variance : Diagnostic de *surajustement* et de *sous-apprentissage*

L'hyperparamètre k influence la complexité de la frontière de décision du classificateur k-NN et, par conséquent, sa capacité de généralisation. En termes généraux, de petites valeurs de k rendent le modèle plus sensible aux échantillons d'entraînement, tandis que des valeurs plus grandes produisent des frontières de décision plus lisses.

Ces comportements sont associés au compromis entre **biais** (*bias*) et **variance** (*variance*). De très petites valeurs de k tendent à augmenter le risque de **surajustement** (*overfitting*), surtout dans des ensembles de

données bruités, alors que des valeurs très grandes peuvent conduire au **sous-apprentissage** (*underfitting*), réduisant la capacité du modèle à capturer les structures locales des données.

Alors que la Figure 8.16 n'a présenté que la précision moyenne obtenue par validation croisée, la Figure 8.17 compare les précisions d'entraînement et de test pour différentes valeurs de k. Les régions mises en évidence sur le graphique représentent le comportement attendu de l'algorithme : un risque plus élevé de surajustement pour de petites valeurs de k, une région intermédiaire qui produit souvent un bon équilibre entre biais et variance, et un risque plus élevé de sous-apprentissage pour de grandes valeurs de k.

Cependant, ces régions doivent être interprétées uniquement comme une référence conceptuelle. Le comportement observé dépend des caractéristiques de l'ensemble de données. Sur la base `load_digits`, par exemple, les images présentent peu de variabilité et une bonne séparation entre les classes, de sorte que de petites valeurs de k peuvent afficher des performances similaires — voire supérieures — aux autres, sans montrer un surajustement significatif.

```
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("Interprétation")
print("- Petites valeurs de k : risque plus élevé de surajustement.")
print("- Valeurs intermédiaires : meilleur compromis entre biais et variance.")
print("- Grandes valeurs de k : risque plus élevé de sous-ajustement.")
print("\nLes régions colorées représentent des tendances générales ;")
print("le comportement observé dépend de l'ensemble de données.")
print(f"\nPrécision maximale sur le test : {maior_acc:.3f}")
print(f"Valeurs de k ayant atteint cette précision : {melhores_k}")
```

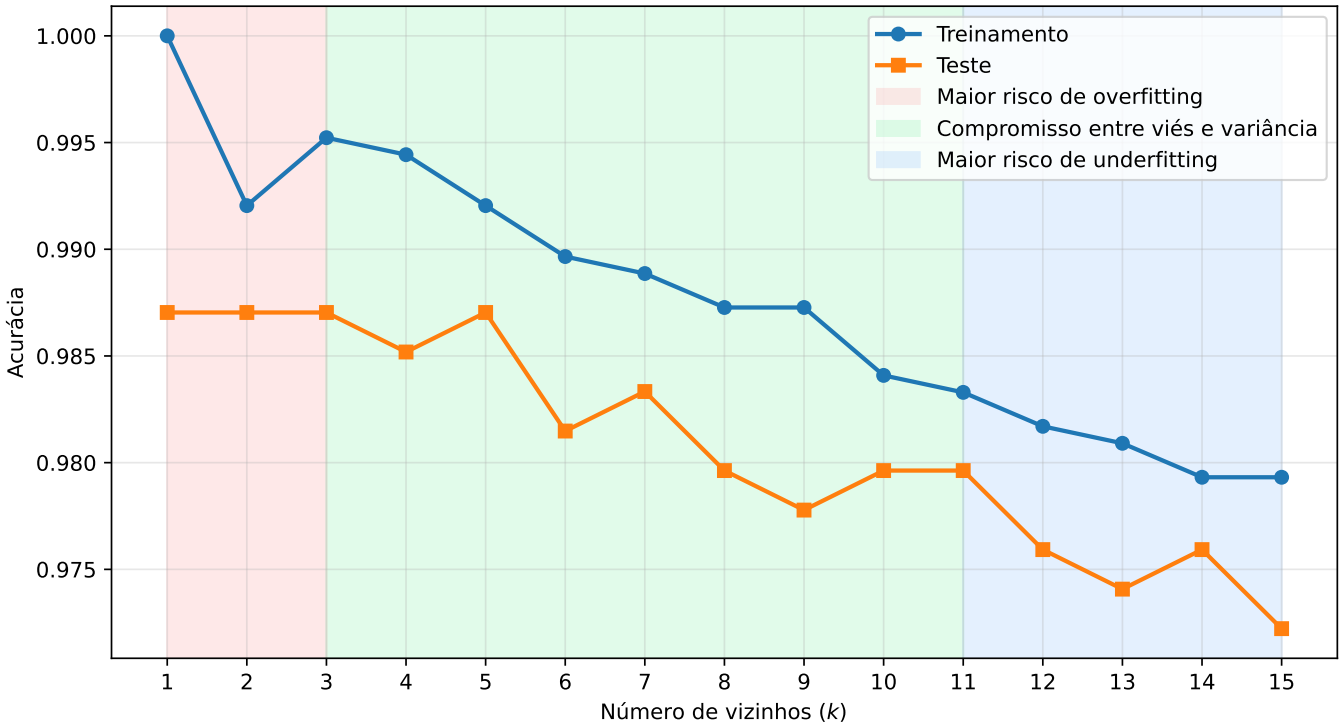


Figure 8.17: Précision sur les ensembles d'entraînement et de test pour différentes valeurs de k. Les régions colorées représentent, de manière conceptuelle, les tendances de comportement du classifieur : risque plus élevé de surajustement (rouge), compromis entre biais et variance (vert) et risque plus élevé de sous-ajustement (bleu).

```
Interprétation
- Petites valeurs de k : risque plus élevé de surajustement.
- Valeurs intermédiaires : meilleur compromis entre biais et variance.
- Grandes valeurs de k : risque plus élevé de sous-ajustement.

Les régions colorées représentent des tendances générales ;
le comportement observé dépend de l'ensemble de données.

Précision maximale sur le test : 0.987
Valeurs de k ayant atteint cette précision : [1, 2, 3, 5]
```

8.13 Projet Pratique 2 : Comparaison de Descripteurs pour la Classification de Textures

Dans le **Chapitre 6**, la variance locale a été utilisée comme descripteur de texture pour **détecter** des anomalies sur des surfaces industrielles, en distinguant des échantillons **conformes** et **défectueux**. Dans ce projet, le problème est reformulé comme une tâche de **classification multiclasse**, dans laquelle différentes représentations de l'image sont utilisées comme entrée pour un classificateur.

Trois types de descripteurs seront considérés : les **intensités des pixels**, le **Local Binary Patterns (LBP)** et l'**Histogram of Oriented Gradients (HOG)**. Pour chaque représentation, un vecteur de caractéristiques sera extrait et servira d'entrée à l'algorithme des k plus proches voisins (k-NN). À la fin, les précisions obtenues par chaque descripteur dans les conditions définies pour cette expérience seront comparées.

L'évaluation sera réalisée par **validation croisée stratifiée en cinq partitions (*5-fold stratified cross-validation*)**. Dans cette procédure, l'ensemble de données est divisé en cinq sous-ensembles en préservant la proportion entre les classes. À chaque itération, une partition est utilisée pour le test et les quatre restantes

pour l'entraînement, le processus étant répété jusqu'à ce que toutes les partitions aient été utilisées comme ensemble de test. À l'issue des cinq exécutions, la précision moyenne et l'écart-type sont calculés pour chaque descripteur.

L'ensemble de données est composé de trois classes de textures synthétiques : **granulaire**, obtenue à partir de bruit gaussien lissé ; **rayée**, formée par des motifs sinusoïdaux périodiques ; et **tachetée**, composée de régions circulaires superposées. Pour introduire de la variabilité entre les échantillons, toutes les images reçoivent une perturbation par bruit gaussien de faible intensité. La Figure 8.18 présente des exemples des trois classes utilisées dans l'expérience.

```
rng = np.random.default_rng(42)

def gerar_textura(classe, tamanho=64, ruido=0.10):
    """Génère une texture synthétique 64x64 appartenant à l'une des trois classes."""
    if classe == "granular":
        escala = rng.uniform(0.14, 0.22)
        img = rng.normal(0.5, escala, (tamanho, tamanho))
        img = cv2.GaussianBlur(img.astype(np.float32), (3, 3), 0)

    elif classe == "listrada":
        n_periodos = rng.uniform(4, 8)
        amplitude = rng.uniform(0.22, 0.38)
        eixo_x = np.linspace(0, n_periodos * np.pi, tamanho)
        base = 0.5 + amplitude * np.sin(eixo_x)
        img = np.tile(base, (tamanho, 1)).astype(np.float32)
        img += rng.normal(0, 0.09, (tamanho, tamanho)).astype(np.float32)

    elif classe == "manchada":
        img = np.full((tamanho, tamanho), 0.5, dtype=np.float32)
        n_manchas = rng.integers(5, 11)
        for _ in range(n_manchas):
            cx, cy = rng.integers(0, tamanho, 2)
            raio = int(rng.integers(3, 11))
            intensidade = float(rng.uniform(0.15, 0.9))
            cv2.circle(img, (int(cx), int(cy)), raio, intensidade, -1)
        img = cv2.GaussianBlur(img, (5, 5), 0)

    else:
        raise ValueError(f"Classe desconhecida: {classe}")

    img = img + rng.normal(0, ruido, (tamanho, tamanho)).astype(np.float32)
    img = np.clip(img, 0, 1)
    return (img * 255).astype(np.uint8)

classes_textura = ["granular", "listrada", "manchada"]
amostras = [gerar_textura(c) for c in classes_textura]

mm.show(amostras, titles=classes_textura, cols=3, figsize=(9, 3))
```

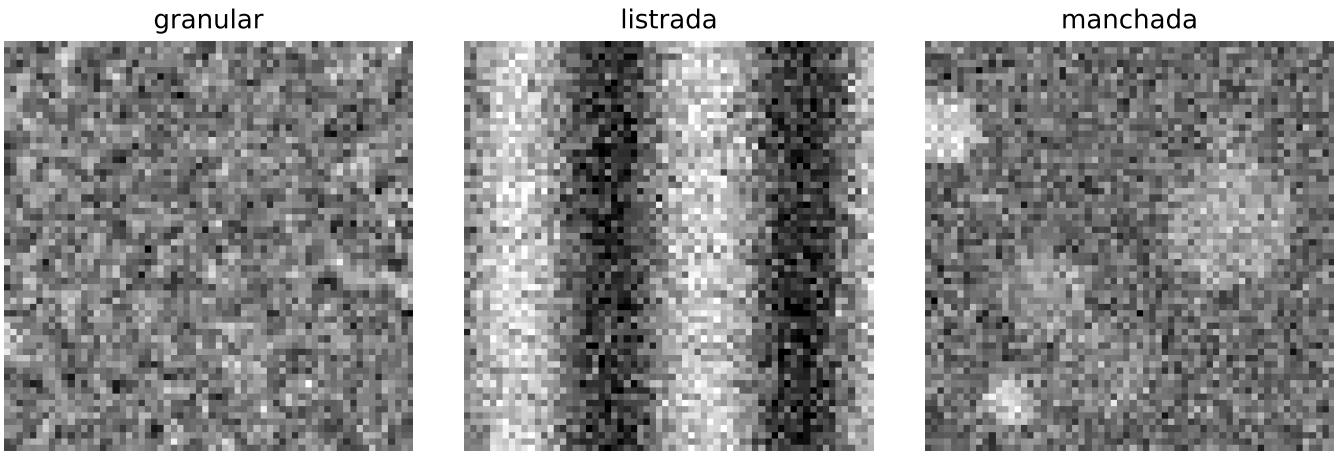


Figure 8.18: Échantillons synthétiques des trois classes de texture utilisées dans l'expérience de classification, générés avec un bruit gaussien et une variabilité intra-classe.

8.13.1 Pipeline d'extraction de caractéristiques et évaluation comparative

Pour comparer les performances des différentes formes de représentation des images, un ensemble de données équilibré contenant 60 échantillons par classe a été généré. À partir de cet ensemble, trois types de vecteurs de caractéristiques ont été extraits, chacun représentant des aspects distincts de l'information visuelle :

1. **Pixels bruts** : vecteur obtenu par l'aplatissement (*flattening*) de la matrice d'intensités de l'image, résultant en un vecteur de $64 \times 64 = 4096$ attributs ;
2. **Histogramme LBP uniforme** : histogramme normalisé des fréquences des motifs locaux produits par l'opérateur LBP uniforme, composé de 10 attributs ;
3. **Descripteur HOG** : vecteur formé par des histogrammes de gradients orientés, qui représentent la distribution spatiale des orientations des contours, totalisant 128 attributs.

Ces descripteurs ayant des échelles et des dimensionalités distinctes, les vecteurs de caractéristiques sont standardisés à l'aide du `StandardScaler`, de sorte que chaque attribut présente une moyenne nulle et un écart-type unitaire. Cette étape évite que des attributs de plus grande amplitude n'influencent de manière disproportionnée le calcul des distances euclidiennes employé par le classificateur.

L'évaluation est réalisée à l'aide de l'algorithme k-NN avec $k = 5$, sous le même protocole de validation croisée stratifiée en cinq partitions décrit dans la section précédente. La précision moyenne obtenue sur les cinq exécutions, voir Figure 8.19, fournit une estimation plus stable des performances du classificateur, réduisant la dépendance à une seule division entre entraînement et test.

```
rng = np.random.default_rng(42)

# 1. Génération de la base de données
X_bruto, X_lbp, X_hog, y_textura = [], [], [], []

for classe in classes_textura:
    for _ in range(60):
        img = gerar_textura(classe, ruido=0.10)

        # Extraction 1 : Pixels bruts
        X_bruto.append(img.ravel())

        # Extraction 2 : Histogramme 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)
```

```
        # Extraction 3 : Descripteur 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. Évaluation statistique via validation croisée 5-fold
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} -> Précision moyenne : {scores.mean():.4f} (± {scores.std():.4f})")

# 3. Tracé du graphique comparatif formel
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   -> Précision moyenne : 0.6889 (± 0.0478)
LBP (Textura)   -> Précision moyenne : 0.7611 (± 0.0648)
HOG (Forma)     -> Précision moyenne : 0.4889 (± 0.0624)
```

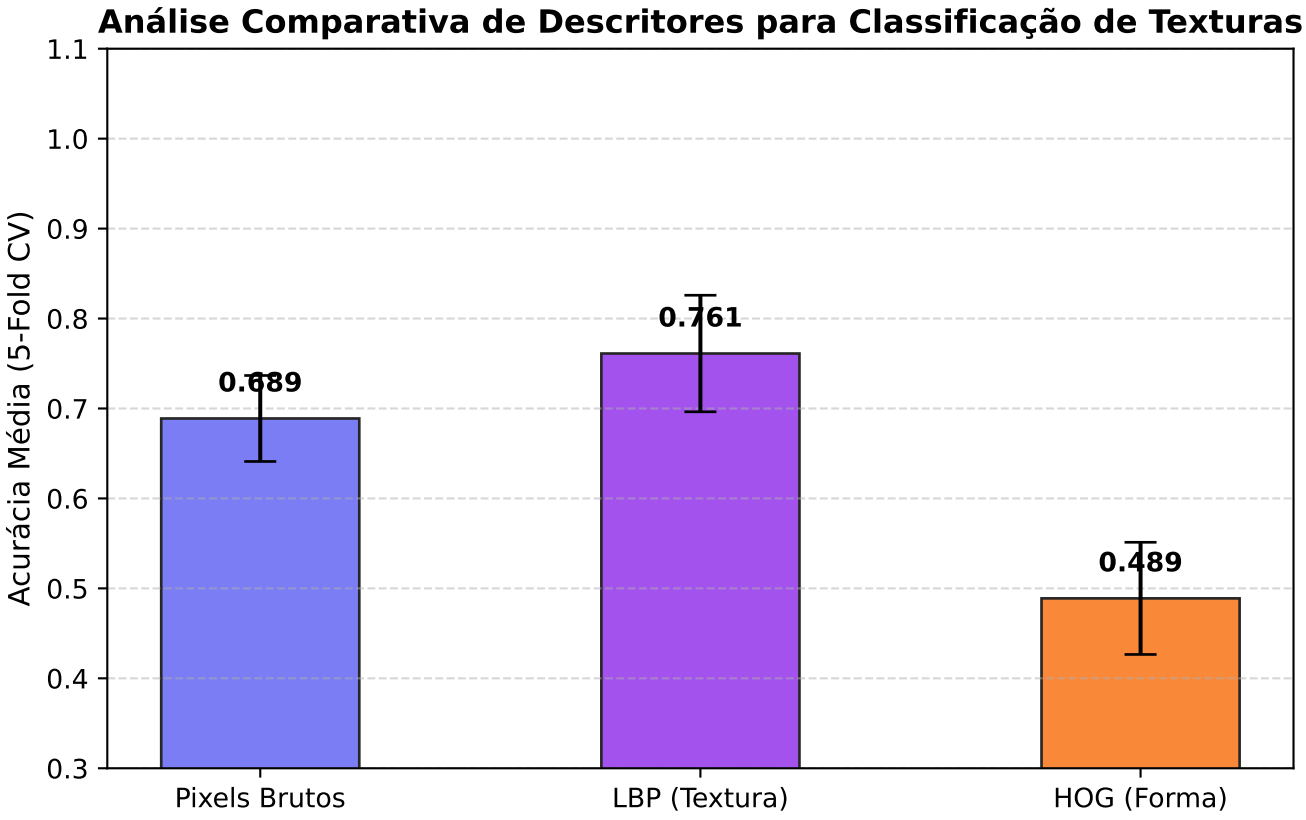


Figure 8.19: Précision moyenne obtenue par validation croisée (5-fold) pour les descripteurs de Pixels Bruts, LBP et HOG appliqués à la base de textures synthétiques.

Pourquoi cela fonctionne-t-il ? — LBP comme représentation des textures

Le descripteur LBP représente la texture d'une image au moyen d'un histogramme normalisé qui comptabilise la fréquence des motifs locaux d'intensité. Plutôt que de stocker directement les valeurs des pixels ou leurs positions, cette représentation résume la distribution des microstructures présentes dans l'image, produisant un vecteur de caractéristiques compact.

Dans cette expérience, trois types de descripteurs ont été comparés : les pixels bruts, LBP et HOG. Les vecteurs formés par les **pixels bruts** préservent toutes les intensités de l'image, mais incorporent également des variations dues au bruit et aux petits déplacements spatiaux, ce qui peut rendre difficile la comparaison entre échantillons au moyen de la distance euclidienne.

Le descripteur **HOG** représente la distribution des orientations des gradients, étant approprié pour décrire les formes et les contours. Comme les images utilisées dans ce projet diffèrent principalement par les propriétés de texture, et non par la présence de contours bien définis, cette représentation tend à capturer moins d'informations discriminatives que le LBP.

Quant au **LBP**, il a été développé spécifiquement pour caractériser les motifs locaux de texture. Son histogramme décrit la fréquence des microstructures présentes dans l'image, indépendamment de leur position exacte, rendant la représentation moins sensible aux petites variations spatiales et aux changements monotones d'éclairage.

Bien que les histogrammes des différentes classes présentent des distributions distinctes, le bruit gaussien introduit lors de la génération des images augmente la variabilité entre les échantillons de la même classe et peut produire des zones de chevauchement dans l'espace des caractéristiques. Par conséquent, certaines textures peuvent être confondues par le classificateur. Néanmoins, lorsque les caractéristiques pertinentes pour distinguer les classes sont associées aux motifs locaux de texture, on s'attend à ce que des descripteurs conçus à cette fin, comme le LBP, produisent des représentations plus informatives que celles basées uniquement sur les intensités des pixels ou les orientations des gradients.

8.13.2 Diagnostic fin du classificateur : précision, rappel et F1-score

L'exactitude résume la performance du classificateur en une seule valeur, mais n'indique pas comment cette performance se répartit entre les différentes classes. Pour une analyse plus détaillée, on utilise des métriques calculées individuellement pour chaque classe.

La **précision** (*precision*) mesure la proportion d'échantillons classés comme appartenant à une classe qui lui appartiennent réellement. Le **rappel** (*recall*) mesure la proportion d'échantillons de la classe qui ont été correctement identifiés par le classificateur. Le **F1-score** correspond à la moyenne harmonique entre précision et rappel, fournissant un indicateur qui équilibre ces deux mesures.

Le rapport indique également le **support** (*support*), c'est-à-dire le nombre d'échantillons de chaque classe présents dans l'ensemble de test. Cette information est importante pour contextualiser les métriques, car les résultats obtenus sur un petit nombre d'échantillons tendent à présenter une plus grande variabilité.

La Figure 8.20 présente ces métriques pour les trois classes de texture. Ensemble, elles permettent d'identifier des différences de performance qui ne sont pas évidentes à partir de la seule exactitude. Par exemple, une classe peut présenter une précision élevée et un rappel plus faible, indiquant que le classificateur commet peu de faux positifs, mais ne parvient pas à identifier une partie des échantillons qui appartiennent réellement à cette classe. Ce type d'analyse aide à comprendre les limites du modèle et à identifier d'éventuelles stratégies pour son amélioration.

```
X_lbp_data = np.array(X_lbp)
y_textura_data = np.array(y_textura)

# Effectuer une division entraînement/test pour le rapport détaillé
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
)

# Mettre à l'échelle les données
scaler = StandardScaler()
Xt_treino_scaled = scaler.fit_transform(Xt_treino)
Xt_teste_scaled = scaler.transform(Xt_teste)

# Entraîner le classificateur 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("=== RAPPORT DE CLASSIFICATION DÉTAILLÉ ===")
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("Interprétation des métriques :")
print("- Précision : parmi les échantillons classés comme appartenant à la classe,",
      "combien étaient corrects ?")
print("- Rappel : parmi les échantillons qui appartiennent réellement à la classe, combien",
      "ont été identifiés ?")
print("- Score F1 : moyenne harmonique entre précision et rappel.")
print("- Support : nombre d'échantillons réels de chaque classe présents dans l'ensemble de
test.")
print("\nLe support ne mesure pas la performance ; il indique simplement combien d'exemples de
chaque",
      "classe ont été \nutilisés dans l'évaluation.")
```

```
=== RAPPORT DE CLASSIFICATION DÉTAILLÉ ===
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
```

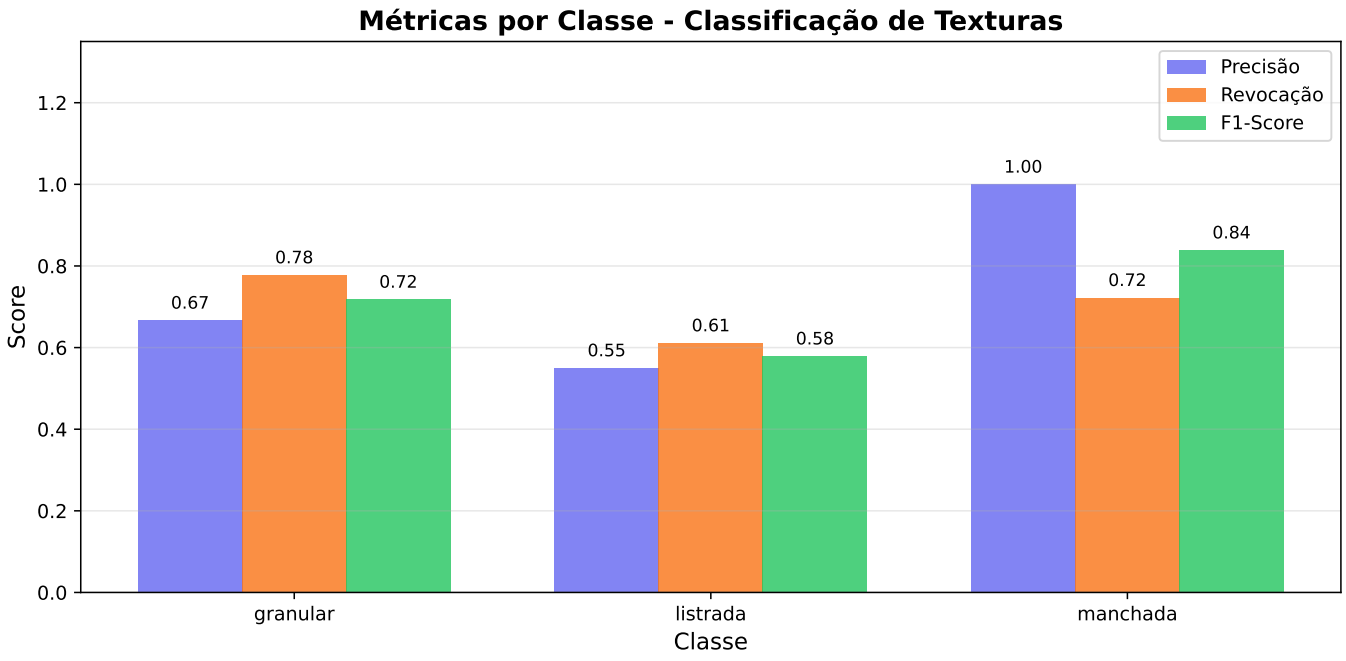


Figure 8.20: Métriques d'évaluation détaillées pour le classificateur k-NN avec descripteurs LBP

```
Interprétation des métriques :
- Précision : parmi les échantillons classés comme appartenant à la classe, combien étaient
corrects ?
- Rappel : parmi les échantillons qui appartiennent réellement à la classe, combien ont été
identifiés ?
- Score F1 : moyenne harmonique entre précision et rappel.
- Support : nombre d'échantillons réels de chaque classe présents dans l'ensemble de test.

Le support ne mesure pas la performance ; il indique simplement combien d'exemples de chaque
classe ont été
utilisés dans l'évaluation.
```

8.14 Limites des descripteurs artisanaux

Les expériences de ce chapitre montrent que les descripteurs classiques peuvent être assez efficaces dans les tâches de classification, mais présentent également des limites importantes :

- **Spécificité :** chaque descripteur a été développé pour représenter un type d'information donné, comme la couleur, la texture ou la forme. Ainsi, un descripteur adapté à une tâche peut ne pas être le plus approprié pour une autre.
- **Dépendance aux hyperparamètres :** la performance de descripteurs comme LBP et HOG dépend du choix de paramètres, tels que le rayon de voisinage, le nombre de points échantillonnés, la taille de la cellule et le nombre d'orientations, qui doivent être ajustés selon l'application.
- **Représentation limitée :** les descripteurs de couleur, de texture et de gradient capturent des propriétés de bas niveau de l'image, mais ne représentent pas directement des concepts sémantiques plus complexes, comme les objets ou les scènes.
- **Malédiction de la dimensionnalité :** des descripteurs trop étendus peuvent réduire l'efficacité des classificateurs basés sur la distance, comme le k-NN.

Ces limites motivent l'évolution des techniques étudiées dans les prochains chapitres. Le **Chapitre 8** présente des méthodes classiques pour la détection et la correspondance de caractéristiques dans les images, tandis que le **Chapitre 9** introduit les **Réseaux de Neurones Convolutionnels**, capables d'apprendre automatiquement des représentations adaptées à chaque tâche à partir des données.

8.15 Résumé

Dans ce chapitre, les fondements de la reconnaissance de formes appliquée aux images ont été présentés. Les principaux concepts étudiés ont été :

- ***Pipeline* de reconnaissance de formes :** acquisition, prétraitement, extraction de descripteurs, classification et évaluation.
- **Descripteurs classiques :** descripteurs de couleur, LBP pour la texture et HOG pour la forme, utilisés pour représenter différentes caractéristiques des images.
- **Normalisation des caractéristiques :** standardisation (*Z-score*) afin d'éviter que les attributs de plus grande magnitude ne dominent le calcul des distances.
- **Classifieur *k*-NN :** classification basée sur les k plus proches voisins dans l'espace des caractéristiques.
- **Choix du paramètre k :** influence de la valeur de k sur les performances du classifieur et utilisation de la validation croisée pour sa sélection.
- **Évaluation des classifieurs :** exactitude, matrice de confusion, précision, rappel et F1-score comme métriques complémentaires de performance.
- **Limites des descripteurs artisanaux :** spécificité, dépendance aux hyperparamètres et difficulté à représenter des informations de haut niveau.

Les concepts ont été illustrés au moyen d'expériences menées sur la base publique `load_digits`, des textures synthétiques générées à des fins pédagogiques et des données simulées de descripteurs de fruits.

8.16 🤖 Utilisation de Gemini Notebook comme tuteur

Dans cette édition, **Gemini Notebook** est présenté comme un outil d'aide à l'étude. Le système utilise exclusivement les documents mis à disposition par l'auteur comme source de connaissances, permettant d'explorer les concepts du chapitre au moyen de questions, de résumés et d'explications liés au matériel étudié.

🎓 Étudiez avec le tuteur intelligent

🚀 ACCÉDER À GEMINI NOTEBOOK : CHAPITRE 07

🌐 **Langue et langage de programmation**

Le projet de ce chapitre dans Gemini Notebook a été construit uniquement avec le texte en **portugais** et les exemples de code en **Python**. Si vous étudiez à partir de l'édition en anglais ou en français, ou si vous suivez le parcours en C++, les réponses du tuteur peuvent ne pas correspondre exactement à la version que vous lisez.

⚠️ **Avis concernant le contenu généré par IA**

Les réponses fournies par Gemini Notebook peuvent contenir des imprécisions ou des omissions. Si nécessaire, confirmez les informations en utilisant le matériel de ce chapitre et d'autres sources académiques fiables. L'exécution des exemples pratiques présentés tout au long du texte reste le meilleur moyen de consolider les concepts étudiés.

8.17 Liste d'exercices

Les exercices suivants explorent et étendent les concepts présentés dans ce chapitre à travers des adaptations des algorithmes implémentés, des analyses expérimentales et des comparaisons entre différentes approches.

1. **(10 %)** Implémentez un descripteur de couleur (histogramme RVB ou TSV, avec au moins 16 *bins* par canal) pour les trois classes de fruits simulés dans la Figure 8.1. Entraînez un classifieur k-NN avec ce

descripteur, comparez sa précision avec celle obtenue par les descripteurs LBP et HOG (Figure 8.9) et discutez dans quelles situations l'information de couleur est la plus discriminative.

2. **(15 %)** Étudiez l'effet de la normalisation des caractéristiques (*Z-score*) sur la performance du k-NN dans un espace d'attributs hétérogène, en combinant des descripteurs de couleur, LBP et HOG dans un seul vecteur. Comparez les résultats obtenus avec et sans normalisation pour au moins trois valeurs de k.

3. **(15 %)** Reproduisez l'analyse de *surapprentissage* et de *sous-apprentissage* de la Figure 8.17 en variant la taille de l'ensemble d'entraînement (par exemple, 20 %, 50 % et 80 % de la base `load_digits`). Discutez comment la quantité d'exemples influence le choix de la valeur de k.

4. **(15 %)** Étendez le Projet Pratique 2 en ajoutant une quatrième classe synthétique de texture. Évaluez la précision, le rappel et le F1-score pour chaque classe, en suivant le modèle de la Figure 8.20, et analysez l'impact de la nouvelle classe sur la matrice de confusion.

5. **(15 %)** Implémentez manuellement le classifieur k-NN, sans utiliser `sklearn` :

   `sklearn.neighbors.KNeighborsClassifier`,

   en complétant la fonction `knn_passo_a_passo` présentée dans le chapitre. Comparez la précision et le temps d'exécution de l'implémentation manuelle avec celle du `scikit-learn` sur des ensembles de données de tailles croissantes et reliez les résultats à la malédiction de la dimensionnalité.

6. **(15 %)** Étudiez l'influence des paramètres `orientations`, `pixels_per_cell` et `cells_per_block` du descripteur HOG sur la base `load_digits`. Évaluez au moins quatre combinaisons de paramètres et discutez du compromis entre la dimensionnalité du descripteur et la performance du classifieur.

7. **(15 %)** Évaluez l'influence des paramètres P (nombre de voisins) et R (rayon) du descripteur LBP sur la classification des textures synthétiques du Projet Pratique 2, en considérant $P \in \{4, 8, 16\}$ et $R \in \{1, 2, 3\}$. Analysez comment ces paramètres affectent la capacité discriminative du descripteur.

8. **(Bonus – 10 %)** Implémentez manuellement la validation croisée *k-fold* pour le classifieur k-NN sur la base `load_digits`, sans utiliser `cross_val_score`, et comparez les résultats avec ceux obtenus par l'implémentation du `scikit-learn` présentée dans la Figure 8.16.

Références du chapitre

La fondation théorique et les expériences présentées dans ce chapitre sont basées sur les références suivantes :

- Gonzalez (2018), pour les fondements des descripteurs statistiques de texture et des opérations de prétraitement appliquées à l'extraction de caractéristiques.
- Szeliski (2022), pour la présentation du pipeline classique de reconnaissance de formes, de l'extraction de descripteurs et de l'évaluation de classificateurs en Vision par Ordinateur.
- Duda (2001), pour les fondements théoriques de la reconnaissance de formes, du classificateur k-NN et de la relation entre biais et variance.
- Cover (1967), pour la formulation originale de l'algorithme des k plus proches voisins.
- Ojala (2002), pour la formulation du descripteur *Local Binary Patterns* (LBP) et de sa variante uniforme, utilisée dans ce chapitre.
- Dalal (2005), pour la formulation du descripteur *Histogram of Oriented Gradients* (HOG), employé dans la représentation de la forme et du contour.
- Pedregosa (2011), pour l'implémentation du classificateur k-NN, des métriques d'évaluation et de la validation croisée dans la bibliothèque `scikit-learn`.
- Quilici-gonzalez (2014), pour la présentation didactique de classificateurs traditionnels de Reconnaissance de Formes, tels que les Arbres de Décision, les Règles de Classification et les Machines à Vecteurs de Support (SVM), complémentaires au classificateur k-NN exploré dans ce chapitre.
- Quilici-gonzalez (2026), pour la mise à jour et l'élargissement de ces contenus dans sa 2e édition, actuellement en production.

Exécuter Colab | Ouvrir GitHub

8.18 💻 Partie Pratique avec des Exercices de Programmation

La présente liste d'exercices de programmation (EP) consolide les formulations théoriques présentées tout au long du Chapitre 7 — Classification d'Images et Reconnaissance de Formes — à travers un parcours pratique appliqué. Contrairement à la manipulation directe des pixels des chapitres précédents, les EP de ce chapitre travaillent avec les **grandeurs intermédiaires** d'un *pipeline* réel de reconnaissance de formes — vecteurs de caractéristiques, distances, étiquettes prédites et réelles, codes binaires locaux et histogrammes d'orientation — permettant de valider manuellement chaque étape du raisonnement sans dépendre de bibliothèques externes d'apprentissage automatique.

L'enchaînement des exercices reproduit le flux conceptuel du chapitre : on commence par l'implémentation manuelle de la règle de décision du classificateur **k-NN** sur un petit espace de caractéristiques ; ensuite, on revisite, sous l'angle de la normalisation des caractéristiques, le classificateur implémenté dans le premier exercice de la liste ; on avance vers le calcul des métriques d'**évaluation** (matrice de confusion, précision et rappel) à partir des étiquettes prédites et réelles ; on poursuit avec le codage manuel du descripteur de texture **LBP** à partir d'un voisinage 3×3 ; on approfondit le calcul de l'histogramme des orientations du descripteur **HOG** pour une seule cellule ; on avance ensuite vers l'intégration de l'**extraction de descripteurs**, de la **classification k-NN** et de l'**évaluation multi-classe** dans un *pipeline* complet de reconnaissance de textures ; et on conclut avec l'application de ce même *pipeline* sur une **image réelle** (format PGM), où le descripteur LBP est calculé directement sur les pixels d'une mosaïque de textures.

🎯 Objectif de ce Carnet

Le carnet permet de développer, valider, organiser et tester des solutions d'**Exercices de Programmation (EPs)** dans des environnements interactifs, comme Colab, avec les mêmes cas de test que Moodle, en les y copiant uniquement au moment d'enregistrer la note officielle.

Téléchargement

Téléchargez `morph.py` et `testsuite.py` en exécutant la cellule ci-dessous :

```
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
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0 | TestSuite : 1.1.2
```

Exécution des tests

Pour évaluer les tests, exécutez `TestSuite("EP07_01.extensão").run()` dans une nouvelle cellule, en remplaçant l'extension par celle du langage utilisé (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). Le système télécharge les cas de test depuis GitHub, exécute le programme et calcule automatiquement la note.

Pour tester directement le code Python, sans enregistrer de fichier, utilisez `run_code(codigo)` en passant le code sous forme de *chaîne de caractères* dans une variable `codigo` :

```
codigo = """
# ... votre code ici ...
"""
TestSuite("EP07_01").run_code(codigo)
```

Résumé des méthodes de `morph.py` (Chap. 7)

La bibliothèque `morph.py` propose deux versions pour la plupart des algorithmes : une version **didactique** (méthodes se terminant par `0`), implémentée pas à pas en NumPy, et une version **classique**, basée sur les bibliothèques `scikit-learn` et `scikit-image`. Les implémentations didactiques sont utilisées dans les **Exercices de Programmation (EPs)**, car elles ne dépendent pas de bibliothèques externes et s'exécutent dans la limite de mémoire de l'environnement **VPL** de Moodle. Les versions classiques, quant à elles, sont plus efficaces et recommandées pour des expériences dans des environnements tels que Colab et Jupyter Notebook, mais elles ne peuvent généralement **pas être utilisées dans les EPs** de Moodle, car la bibliothèque `scikit-learn` dépasse la mémoire disponible dans le VPL.

1. **Lecture des données (`readClasses`, `readDataset`, `readTrain`, `readTest`)**
   Elles normalisent l'entrée des ensembles d'entraînement et de test, en retournant les matrices de caractéristiques (X) et les vecteurs d'étiquettes (y).

2. **Classification (`knn0` / `knn`)**
   Elles implémentent l'algorithme des **k-plus proches voisins (k-NN)** pour la classification binaire et multiclasse, en utilisant la distance euclidienne ou de Manhattan.

3. **Normalisation (`zscore0` / `zscore`)**
   Elles appliquent la normalisation *z-score* aux attributs, réduisant les différences d'échelle avant la classification.

4. **Évaluation (`confusion0` / `confusion`)**
   Elles calculent la matrice de confusion et des métriques telles que l'exactitude, la précision et le rappel, tant pour les problèmes binaires que multiclasses.

5. **Descripteur de texture (`lbp0` / `lbp`)**
   Elles calculent le ***Local Binary Pattern* (LBP)**, permettant d'obtenir la carte LBP, le code d'un pixel ou l'histogramme d'une région de l'image.

6. **Descripteur de forme (`hog0` / `hog`)**
   Elles calculent le ***Histogram of Oriented Gradients* (HOG)**, produisant des histogrammes des orientations des gradients pour représenter les informations de forme et de contour.

8.18.1 EP07_01 🟢 Classifieur k-NN Pas à Pas

Le `KNeighborsClassifier` de `scikit-learn`, utilisé tout au long du chapitre, dissimule derrière un simple appel (`.fit` / `.predict`) une règle de décision très simple : pour chaque nouvelle observation, calculer la distance à tous les exemples d'entraînement, sélectionner les k plus proches et voter pour la classe majoritaire parmi eux.

Avant de vous fier à la bibliothèque, vous êtes chargé d'implémenter cette règle de zéro, pour un espace de caractéristiques bidimensionnel, exactement comme le simulateur interactif de frontière de décision du chapitre le fait en interne à chaque clic de l'utilisateur.

8.18.1.1 📋 Directives d'implémentation

1. **Quantité et paramètre :** Lire l'entier N (nombre d'exemples d'entraînement) et l'entier impair k (nombre de voisins).
2. **Exemples d'entraînement :** Pour chacun des N exemples, lire trois valeurs : les coordonnées x et y (réelles) et l'étiquette r (entier, 0 ou 1).

3. **Requêtes :** Lire l'entier Q (nombre de points de requête) puis les coordonnées x_q, y_q (réelles) de chaque requête.
4. **Distance :** Pour chaque requête, calculer la distance euclidienne à **tous** les exemples d'entraînement :

$$d(x_q, x_i) = \sqrt{(x_q - x_i)^2 + (y_q - y_i)^2}.$$

5. **Sélection des voisins :** Trier les exemples par distance croissante et sélectionner les k premiers. En cas d'**égalité de distance** à la frontière du k-ième voisin, départager par l'exemple lu **en premier** dans l'entrée (ordre de lecture stable).
6. **Vote majoritaire :** Compter les votes de chaque classe parmi les k voisins sélectionnés. S'il y a **égalité dans le vote** (seulement possible lorsque k est pair, ce qui ne devrait pas se produire selon la directive du point 1, mais traitez-le défensivement), attribuez la classe du voisin le plus proche parmi les classes à égalité.
7. **Sortie :** Pour chaque requête, dans l'ordre d'entrée, imprimer la classe prédite. À la fin, imprimer le total de requêtes classées comme classe `1`.

8.18.1.2 📌 Contraintes computationnelles

- **Métrique fixe :** utilisez exclusivement la distance euclidienne (pas la *distance au carré*) pour le tri, bien que le résultat de la comparaison soit le même.
- **k toujours impair :** l'entrée garantit k impair et $k \leq N$; néanmoins, implémentez le départage du point 6 par robustesse.
- **Stabilité :** lors du tri par distance, préservez l'ordre relatif des exemples ayant la même distance (tri stable).

8.18.1.3 🧠 Fondement théorique

Élément	Rôle dans le k-NN
Espace de caractéristiques	Ensemble de tous les vecteurs (x, y) possibles
Distance euclidienne	Mesure de similarité entre observations
k petit	Frontière irrégulière, variance élevée
k grand	Frontière lisse, biais élevé
Vote majoritaire	Règle de décision $\hat{y} = \text{moda}\{y_i : x_i \in N_k(x)\}$

8.18.1.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

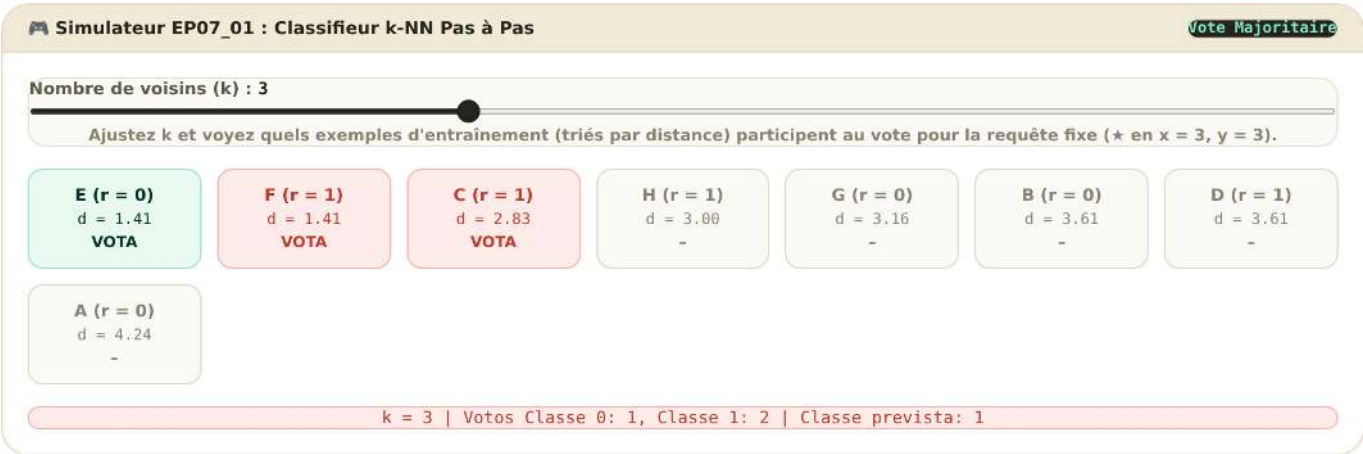
- Ligne 1 : entiers N et k, séparés par un espace.
- Les N lignes suivantes : trois valeurs par ligne — x, y (réelles) et r (entier $\in \{0, 1\}$), séparées par un espace.
- Ligne suivante : entier Q.
- Les Q lignes suivantes : deux valeurs par ligne — x_q, y_q (réelles), séparées par un espace.

Sortie :

- Q lignes, chacune avec la classe prédite (`0` ou `1`) pour la requête respective, dans l'ordre d'entrée.
- Dernière ligne : `Total classe 1 : X`.

8.18.1.5 📌 Exemples

Entrée	Sortie	Observation
4 3 0 0 0 1 0 0 5 5 1 6 5 1 1 1 1	0 Total classe 1 : 0	Requête proche du groupe de classe 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 classe 1 : 1	Avec $k = 1$, chaque requête hérite de la classe du voisin le plus proche.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap07/sim-ep0701.html

Figure 8.21: Simulateur EP07_01 : Classifieur k-NN Pas à Pas

```
%%writefile EP07_01.py
# Code Python
```

```
Overwriting EP07_01.py
```

```
TestSuite("EP07_01.py").run()
```

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

8.18.2 EP07_02 🟡 Normalisation *Z-score* et Robustesse du k-NN à des Échelles Distinctes

Cet exercice revisite le classificateur implémenté dans l'**EP07_01**, cette fois sous l'angle discuté dans la section *L'Impact de l'Échelle et la Normalisation des Caractéristiques* du chapitre : le k-NN décide en se basant sur la distance entre les vecteurs, de sorte qu'une caractéristique mesurée sur une échelle beaucoup plus grande que les autres tend à **dominer** le calcul de la distance, même lorsqu'elle n'est pas la plus pertinente pour séparer les classes.

Un système d'inspection enregistre, pour chaque pièce, sa **superficie** (en pixels, pouvant atteindre des centaines ou des milliers) et sa **circularité** (toujours entre 0 et 1). Vous êtes chargé de classer de nouvelles

pièces par k-NN de deux manières — avec et sans la standardisation *Z-score* présentée dans le chapitre — et de rapporter dans quels cas les deux approches **divergent**.

8.18.2.1 📋 Directives d'Implémentation

1. **Quantité et paramètre :** Lire l'entier N (nombre d'exemples d'entraînement) et l'entier impair k.
2. **Exemples d'entraînement :** Pour chacun des N exemples, lire trois valeurs : la superficie x_1 (réelle), la circularité x_2 (réelle) et l'étiquette r (entier, 0 ou 1).
3. **Requêtes :** Lire l'entier Q puis les coordonnées x_1, x_2 de chaque requête.
4. **Classification sans normalisation :** Pour chaque requête, classifiez-la par k-NN directement sur (x_1, x_2), avec la distance euclidienne et les mêmes règles de départage que l'EP07_01 (ordre de lecture pour les distances à égalité ; voisin le plus proche entre les classes à égalité lors du vote).
5. **Paramètres de normalisation :** Calculer la moyenne μ_j et l'écart-type **populationnel** σ_j (division par N, non par $N-1$ — la même convention adoptée par la classe `StandardScaler`) de chaque caractéristique $j \in \{1, 2\}$, **exclusivement sur l'ensemble d'entraînement**.
6. **Standardisation :** Transformez chaque caractéristique d'entraînement et de requête par
$$z_j = \frac{x_j - \mu_j}{\sigma_j}.$$
Si $\sigma_j = 0$ (caractéristique constante dans l'entraînement), définissez $z_j = 0$ pour tous les échantillons de cette caractéristique, évitant ainsi la division par zéro.
7. **Classification avec normalisation :** Répétez la classification k-NN du point 4, désormais sur les vecteurs standardisés (z_1, z_2), avec les mêmes règles de départage.
8. **Sortie :** Pour chaque requête, dans l'ordre d'entrée, imprimer les deux classes prédites. À la fin, imprimer le nombre de requêtes où les deux classifications **divergent**.

8.18.2.2 📌 Contraintes Computationnelles

- **Ajustement uniquement sur l'entraînement :** μ_j et σ_j sont calculés uniquement à partir de l'ensemble d'entraînement et réappliqués aux requêtes — jamais recalculés à partir de celles-ci. Cette pratique évite la **fuite de données** (*data leakage*), mentionnée dans la section sur la normalisation du chapitre.
- **Écart-type populationnel :** utilisez $\sigma_j = \sqrt{\frac{1}{N}\sum_i (x_{i,j} - \mu_j)^2}$, et non la version échantillonnale (division par $N-1$).
- **Caractéristique constante :** traitez $\sigma_j = 0$ comme un cas spécial (point 6) ; aucune erreur de division par zéro ne doit survenir.
- **Règles de départage :** réutilisez exactement les conventions de l'EP07_01, tant pour la sélection des k voisins que pour le vote majoritaire.

8.18.2.3 🧠 Fondements Théoriques

Élément	Rôle
Standardisation *Z-score*	Rééchelonne chaque caractéristique pour une moyenne de 0 et un écart-type de 1, rendant les échelles hétérogènes comparables
Ajustement (*fit*) uniquement sur l'entraînement	Garantit que l'évaluation sur les requêtes reflète uniquement ce que le modèle a appris lors de l'entraînement
Distance euclidienne sans normalisation	Dominée par la caractéristique de plus grande amplitude — ici, la superficie
Prédiction divergente	Met en évidence que l'échelle des caractéristiques, et pas seulement l'algorithme ou les données, peut déterminer la frontière de décision du k-NN

Cet exercice renforce, de manière contrôlée, la raison pour laquelle le `StandardScaler` est appliqué avant le k-NN tout au long du chapitre : sans cette étape, les caractéristiques de circularité — même étant hautement discriminatives — peuvent être pratiquement ignorées par le classificateur face à une caractéristique de superficie avec une amplitude des centaines de fois plus grande.

8.18.2.4 📦 Spécification d'Entrée et de Sortie (VPL)

Entrée :

- Ligne 1 : Entiers N et k, séparés par un espace.
- Les N lignes suivantes : trois valeurs par ligne — x_1, x_2 (réelles) et r (entier $\in \{0, 1\}$), séparées par un espace.
- Ligne suivante : entier Q.
- Les Q lignes suivantes : deux valeurs par ligne — x_1, x_2 (réelles) de la requête, séparées par un espace.

Sortie :

- Q lignes, au format `SemNorm=<0|1> ComNorm=<0|1>`, dans l'ordre d'entrée des requêtes.
- Dernière ligne : `Divergiu: <int>`.

8.18.2.5 📌 Exemples

Entrée	Sortie	Observation
4 3 10 0.9 0 12 0.85 0 900 0.2 1 950 0.25 1 1 500 0.88	SemNorm=1 ComNorm=0 Divergiu: 1	Sans normalisation, la superficie (échelle des centaines) domine la distance et la requête est classée comme classe `1`. Après standardisation, la circularité — beaucoup plus proche des échantillons de classe `0` — commence à peser de manière comparable, et la prédiction change pour `0`.
2 1 0 0.5 0 100 0.5 1 1 60 0.5	SemNorm=1 ComNorm=1 Divergiu: 0	La circularité est constante dans l'entraînement ($\sigma_2 = 0$) ; selon la règle du point 6, $z_2 = 0$ pour tous les échantillons, et la classification ne dépend que de la superficie dans les deux cas.

```
%%writefile EP07_02.py
# Code Python
```

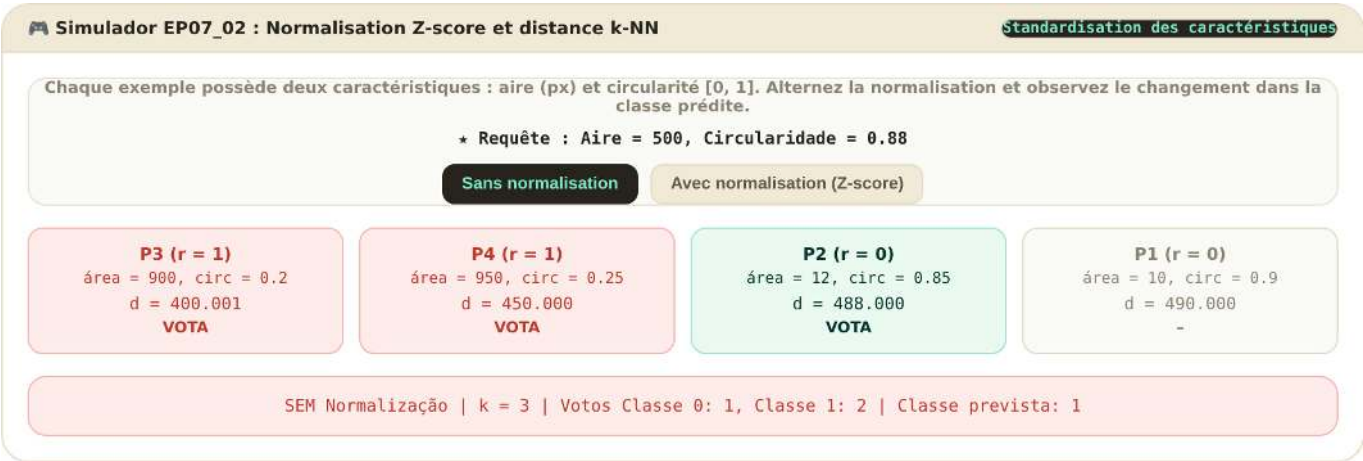
```
Overwriting EP07_02.py
```

```
TestSuite("EP07_02.py").run()
```

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

8.18.3 EP07_03 🟡 Évaluation par Matrice de Confusion

Un classifieur binaire de qualité de soudure a été entraîné et testé sur une ligne de production. Pour chaque pièce inspectée, le système a enregistré le label **réel** (obtenu par un expert) et le label **prédit** par le classifieur, où `1` représente « défectueuse » et `0` représente « conforme ».



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap07/sim-ep0702.html

Figure 8.22: Simulateur EP07_02: Effet de la Normalisation *Z-score* sur la Distance k-NN

La direction qualité souhaite connaître non seulement l'exactitude du système, mais aussi sa **précision** (lorsque le système signale un défaut, à quelle fréquence a-t-il raison ?) et son **rappel** (parmi toutes les pièces réellement défectueuses, combien le système a-t-il réussi à identifier ?) — la distinction discutée dans la section d'évaluation des classifieurs du chapitre.

8.18.3.1 📋 Directives d'Implémentation

1. **Quantité :** Lire l'entier N (nombre de pièces inspectées).
2. **Données de chaque pièce :** Pour chacune des N pièces, lire deux entiers — le label réel y et le label prédit $\hat{y}$ (tous deux $\in \{0, 1\}$).
3. **Matrice de confusion :** En considérant la classe `1` (défectueuse) comme **positive**, compter :
   - VP (Vrai Positif) : $y = 1$ et $\hat{y} = 1$;
   - FP (Faux Positif) : $y = 0$ et $\hat{y} = 1$;
   - FN (Faux Négatif) : $y = 1$ et $\hat{y} = 0$;
   - VN (Vrai Négatif) : $y = 0$ et $\hat{y} = 0$.
4. **Métriques :** Calculer

$$\text{Exactitude} = \frac{VP + VN}{N}, \quad \text{Précision} = \frac{VP}{VP + FP}, \quad \text{Rappel} = \frac{VP}{VP + FN}.$$

5. **Cas dégénérés :** Si $VP + FP = 0$ (aucune prédiction positive), afficher `Precisao: indefinida`. Si $VP + FN = 0$ (aucun cas positif réel), afficher `Revocacao: indefinida`.
6. **Arrondi :** Toutes les métriques numériques doivent être arrondies à 4 décimales (*round half away from zero*) uniquement lors de l'affichage.

8.18.3.2 📌 Contraintes Computationnelles

- **Convention de classe positive fixe :** la classe `1` est toujours la classe positive dans cet exercice, indépendamment de sa fréquence relative.
- **Protection contre la division par zéro :** implémentez les cas dégénérés du point 5 avant d'effectuer la division.
- **Ordre de sortie :** suivez exactement l'ordre spécifié dans la section de sortie, même dans les cas dégénérés.

8.18.3.3 🧠 Fondement Théorique

Métrique	Question à laquelle elle répond	Sensible au déséquilibre ?
Exactitude	Quelle fraction des pièces a été classée correctement ?	Oui — peut masquer des erreurs dans la classe minoritaire
Précision	Parmi les pièces signalées comme défectueuses, combien le sont réellement ?	Pénalise les faux positifs
Rappel	Parmi les pièces réellement défectueuses, combien ont été détectées ?	Pénalise les faux négatifs

Dans un contexte industriel, un **rappel** faible est souvent plus grave qu'une **précision** faible : laisser passer une pièce défectueuse (faux négatif) tend à être plus coûteux que d'inspecter manuellement une bonne pièce signalée par erreur (faux positif).

8.18.3.4 📦 Spécification d'Entrée et de Sortie (VPL)

Entrée :

- Ligne 1 : Entier N.
- Les N lignes suivantes : deux entiers par ligne — y et $\hat{y}$, séparés par un espace.

Sortie (dans cet ordre exact) :

```
VP=<int> FP=<int> FN=<int> VN=<int>
Acuracia: <valeur ou métrique indéfinie>
Precisao: <valeur ou indéfinie>
Revocacao: <valeur ou indéfinie>
```

8.18.3.5 📌 Exemples

Entrée	Sortie	Observation
4 1 1 0 1 1 0 0 0	VP=1 FP=1 FN=1 VN=1 Acuracia: 0.5000 Precisao: 0.5000 Revocacao: 0.5000	Une erreur de chaque type.
3 0 0 0 0 0 0	VP=0 FP=0 FN=0 VN=3 Acuracia: 1.0000 Precisao: indefinida Revocacao: indefinida	Aucun cas positif réel ni prédit.

```
%%writefile EP07_03.py
# Code Python
```

```
Overwriting EP07_03.py
```

```
TestSuite("EP07_03.py").run()
```

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

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap07/sim-ep0703.html

Figure 8.23: Simulateur EP07_03 : Précision x Rappel

8.18.4 EP07_04 🟠 Codage Manuel du Descripteur LBP

La fonction `local_binary_pattern` de `scikit-image`, utilisée dans le projet de classification de textures, calcule automatiquement le code LBP de chaque pixel d'une image. Avant de l'utiliser comme une boîte noire, vous avez été chargé d'implémenter manuellement le calcul du code LBP classique ($P = 8$, $R = 1$) pour le pixel central d'un voisinage 3×3, exactement comme défini dans l'équation du chapitre.

En plus du code, le système d'inspection de textures doit également savoir si ce motif est **uniforme** — un motif est uniforme lorsque le nombre de transitions ($0 \to 1$ ou $1 \to 0$) en parcourant les 8 bits **circulairement** (en revenant du dernier bit au premier) est **au plus 2**, propriété exploitée par la variante *uniforme* du LBP mentionnée dans le chapitre.

8.18.4.1 📋 Directives d'Implémentation

1. **Quantité :** Lire l'entier T (nombre de voisinages à traiter).
2. **Données de chaque voisinage :** Pour chacun des T voisinages, lire une matrice 3×3 d'entiers (intensités), fournie en 3 lignes de 3 valeurs chacune. Le pixel central est la position `[1][1]`.
3. **Ordre des voisins :** Parcourir les 8 voisins dans le sens **horaire**, en commençant par le coin supérieur gauche, dans l'ordre suivant des positions `[ligne][colonne]` : `[0][0]`, `[0][1]`, `[0][2]`, `[1][2]`, `[2][2]`, `[2][1]`, `[2][0]`, `[1][0]`. C'est l'indice $p = 0, 1, \ldots, 7$ de l'équation du LBP.
4. **Fonction seuil :** Pour chaque voisin p avec une intensité g_p et un centre g_c, calculer $s(g_p - g_c)$, qui vaut `1` si $g_p \geq g_c$ et `0` sinon.
5. **Code LBP :** Calculer
$$\mathrm{LBP} = \sum_{p=0}^{7} s(g_p - g_c)\, 2^p.$$
6. **Transitions :** En considérant la séquence circulaire de bits $s_0, s_1, \ldots, s_7$ (dans l'ordre du point 3), compter combien de paires consécutives **adjacentes dans la séquence circulaire** (y compris la paire s_7, s_0) diffèrent entre elles.
7. **Classification :** Si le nombre de transitions est ≤ 2, classer comme `UNIFORME` ; sinon, `NAO_UNIFORME`.
8. **Sortie :** Pour chaque voisinage, dans l'ordre d'entrée, imprimer le code LBP (entier décimal, 0–255), le nombre de transitions et la classification.

8.18.4.2 📌 Contraintes Computationnelles

- **Ordre fixe des voisins :** l'ordre du point 3 est obligatoire — l'inverser produit un code numériquement différent, même en représentant le même motif visuel.
- **Comparaison non stricte :** $s(z) = 1$ lorsque $z \geq 0$ (le chapitre lui-même définit l'égalité comme incluse dans le cas `1`).
- **Comptage circulaire :** ne pas oublier la paire qui ferme le cycle (s_7 avec s_0) ; ignorer cette paire est une erreur courante qui classe incorrectement les motifs uniformes.

8.18.4.3 🧠 Fondement Théorique

Motif (bits $s_0 \dots s_7$)	Transitions	Interprétation
`00000000` ou `11111111`	0	Région homogène (tache claire ou sombre)
`00001111`	2	Bord simple entre deux régions
`01010101`	8	Texture de contraste alterné — non uniforme

Les motifs uniformes se concentrent dans les régions de texture lisse ou de bords simples ; les motifs non uniformes tendent à correspondre à du bruit haute fréquence. C'est pourquoi l'histogramme LBP *uniforme*, utilisé dans le projet de classification de textures, regroupe tous les motifs non uniformes dans un seul compartiment, réduisant la dimensionnalité du descripteur.

8.18.4.4 📦 Spécification d'Entrée et de Sortie (VPL)

Entrée :

- Ligne 1 : Entier T.
- Pour chaque voisinage : 3 lignes avec 3 entiers chacune (matrice 3×3).

Sortie :

- T lignes, au format `LBP=<int> transicoes=<int> <UNIFORME|NAO_UNIFORME>`.

8.18.4.5 📌 Exemples

Entrée	Sortie	Observation
1 10 10 10 10 50 10 10 10 10	LBP=0 transicoes=0 UNIFORME	Le centre est le plus clair ; tous les voisins génèrent un bit 0.
1 90 90 90 10 50 10 90 90 90	LBP=119 transicoes=4 NAO_UNIFORME	Voisins clairs et sombres alternés dans le voisinage.

```
%%writefile EP07_04.py
# Code Python
```

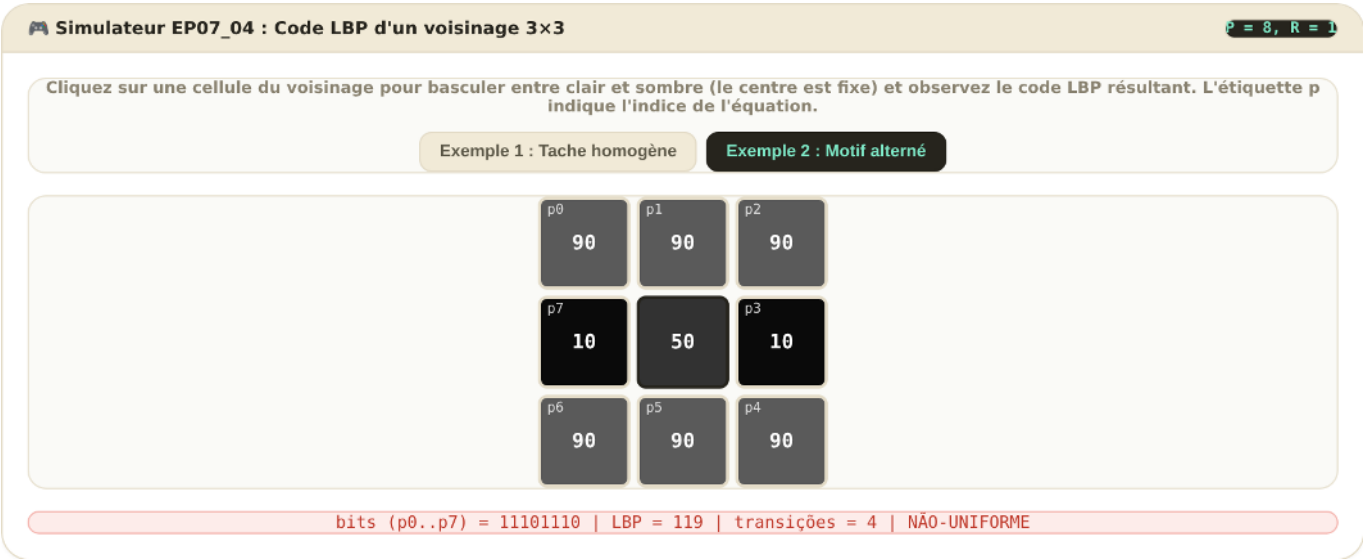
```
Overwriting EP07_04.py
```

```
TestSuite("EP07_04.py").run()
```

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

8.18.5 EP07_05 🔴 Histogramme des orientations d'une cellule HOG

La fonction `hog` de `scikit-image`, utilisée dans le projet de classification de chiffres, divise l'image en petites **cellules** et, pour chacune, construit un histogramme des orientations du gradient pondéré par la magnitude — exactement l'étape centrale décrite dans la section sur le descripteur HOG du chapitre.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap07/sim-ep0704.html

Figure 8.24: Simulateur EP07_04 : Code LBP d'un voisinage 3×3

Vous êtes chargé d'implémenter ce calcul pour une seule cellule, à partir des valeurs de magnitude et d'orientation du gradient **déjà calculées** pour chaque pixel de la cellule (ce qui évite le calcul des dérivées partielles).

8.18.5.1 📋 Directives d'implémentation

1. **Dimensions :** Lire les entiers n (la cellule a $n \times n$ pixels) et B (nombre de compartiments de l'histogramme).
2. **Magnitudes :** Lire n lignes avec chacune n valeurs réelles, représentant $|\nabla f(x, y)|$ pour chaque pixel de la cellule.
3. **Orientations :** Lire encore n lignes avec chacune n valeurs réelles, représentant $\theta(x, y)$ en **degrés**, déjà converties dans l'intervalle **non signé** $[0^\circ, 180^\circ)$, comme conventionnellement utilisé par le HOG.
4. **Compartiments :** Les B compartiments couvrent $[0^\circ, 180^\circ)$ en bandes égales de largeur $180/B$ degrés. Un pixel avec une orientation θ appartient au compartiment $\lfloor \theta/(180/B) \rfloor$; si cet indice est égal à B (possible uniquement lorsque θ est exactement 180°, ce qui ne devrait pas se produire selon la directive du point 3), utilisez le compartiment $B - 1$.
5. **Histogramme brut :** Pour chaque pixel, accumulez sa **magnitude** (et non son compte) dans le compartiment correspondant :

$$H[b] = \sum_{(x,y)\,:\,\mathrm{bin}(\theta(x,y))=b} |\nabla f(x, y)|.$$

6. **Normalisation L2 :** Après avoir construit H, normalisez-le pour obtenir $\hat{H}$:

$$\hat{H}[b] = \frac{H[b]}{\sqrt{\sum_{j=0}^{B-1} H[j]^2 + \epsilon}}, \qquad \epsilon = 10^{-6}.$$

7. **Sortie :** Imprimez l'histogramme brut H (arrondi à 2 décimales) sur une ligne, suivi de l'histogramme normalisé $\hat{H}$ (arrondi à 4 décimales) sur une autre ligne, tous deux avec les B valeurs séparées par des espaces, dans l'ordre des compartiments.

8.18.5.2 📌 Contraintes computationnelles

- ***Binning* non signé :** l'intervalle des orientations est $[0, 180)$, pas $[0, 360)$ — les gradients dans des directions opposées (différence de 180°) contribuent au **même** compartiment, convention standard du HOG pour la détection d'objets.

- **Accumulation par magnitude, pas par compte :** l'histogramme pondère chaque pixel par sa magnitude de gradient, il ne compte pas simplement combien de pixels tombent dans chaque compartiment.
- **Constante de stabilisation :** le $\epsilon = 10^{-6}$ dans le dénominateur de la normalisation évite la division par zéro lorsque la cellule est complètement homogène (toutes les magnitudes nulles).

8.18.5.3 📐 D'où proviennent les matrices d'entrée

Avant cet EP, chaque pixel (x, y) de l'image passe par :

$$G_x = f(x+1, y) - f(x-1, y), \qquad G_y = f(x, y+1) - f(x, y-1)$$

$$|\nabla f| = \sqrt{G_x^2 + G_y^2}, \qquad \theta_{\text{signé}} = \operatorname{atan2}(G_y, G_x)$$

Comme le HOG ignore la polarité du contraste, l'angle est replié dans l'intervalle non signé :

$$\theta = \theta_{\text{signé}} \bmod 180^\circ$$

En répétant cela pour tous les pixels d'une cellule $n \times n$, on obtient les deux matrices d'entrée de cet exercice : les **magnitudes** $|\nabla f|$ et les **orientations** $\theta \in [0^\circ, 180^\circ)$.

8.18.5.4 🧠 Fondement théorique

Étape	Rôle
Magnitude du gradient	Pondère la contribution de chaque pixel — les bords forts pèsent plus que le bruit faible
Orientation non signée	Rend le descripteur invariant à la polarité du contraste (clair→sombre vs. sombre→clair)
Histogramme par cellule	Résume la distribution locale des bords en un vecteur compact
Normalisation L2	Réduit la sensibilité du descripteur aux variations globales d'éclairage et de contraste

La concaténation des histogrammes normalisés de toutes les cellules de l'image — non implémentée dans cet exercice — forme le vecteur de caractéristiques HOG complet, utilisé comme entrée du classifieur k-NN dans le projet du chapitre.

8.18.5.5 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : entiers n et B.
- Les n lignes suivantes : n magnitudes réelles chacune.
- Les n lignes suivantes : n orientations réelles (degrés, $[0, 180)$) chacune.

Sortie :

- Ligne 1 : les B valeurs de l'histogramme brut, arrondies à 2 décimales.
- Ligne 2 : les B valeurs de l'histogramme normalisé, arrondies à 4 décimales.

8.18.5.6 📌 Exemples

Entrée	Sortie	Observation
2 2 1.0 2.0 3.0 4.0 10 100 170 20	5.00 5.00 0.7071 0.7071	Bin de largeur 90° : $[0, 90)$ et $[90, 180)$; les magnitudes 1 et 4 tombent dans le bin 0, 2 et 3 dans le 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	Cellule homogène : ϵ évite la division par zéro.

```
%%writefile EP07_05.py
# Code Python
```

```
Overwriting EP07_05.py
```

```
TestSuite("EP07_05.py").run()
```

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

8.18.6 EP07_06 🟣 *Pipeline* complet : Descripteurs + k-NN + Évaluation multi-classe

Cet exercice intègre les trois étapes centrales du chapitre dans un *pipeline* unique, reproduisant en miniature le **Projet pratique 2** (classification de textures synthétiques par LBP) : un ensemble d'histogrammes de descripteurs **déjà extraits** (comme s'il s'agissait d'histogrammes LBP) est utilisé pour entraîner un classifieur k-NN, qui est à son tour évalué sur un ensemble de test indépendant au moyen d'une matrice de confusion multi-classe.

Contrairement à l'EP07_01, ici l'espace des caractéristiques a une dimension arbitraire H (la taille de l'histogramme), il existe plus de deux classes, et la métrique de distance est un paramètre d'entrée — ce qui permet de reproduire l'expérience de comparaison de métriques discutée dans le chapitre.

8.18.6.1 📋 Directives d'implémentation

1. **Classes :** Lire l'entier C (nombre de classes) suivi de C noms de classe (*strings* sans espace), dans l'ordre où ils doivent apparaître dans la matrice de confusion.
2. **Configuration :** Lire l'entier H (dimension des histogrammes), la *string* M (métrique : `euclidiana` ou `manhattan`) et l'entier impair k.
3. **Entraînement :** Lire l'entier N puis N lignes, chacune contenant le nom de la classe suivi de H valeurs réelles (l'histogramme du descripteur).
4. **Test :** Lire l'entier Q puis Q lignes, chacune contenant le nom de la classe **réelle** suivi de H valeurs réelles (l'histogramme du descripteur de l'échantillon de test).
5. **Distance :** Pour chaque échantillon de test, calculer la distance à chaque exemple d'entraînement en utilisant la métrique 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. **Classification k-NN :** Sélectionner les k exemples d'entraînement les plus proches (départage des distances par l'ordre de lecture, comme dans l'EP07_01) et classer par la classe majoritaire parmi eux. En cas **d'égalité de vote** entre deux ou plusieurs classes, choisir celle qui apparaît **en premier** dans la liste des classes du point 1.

🎮 **Simulateur EP07_05 : Histogramme d'orientations d'une cellule** — cellule 3×3 fixe

Ajustez B et suivez comment la **matrice d'orientations** (indépendante de celle des magnitudes) est mappée vers les compartiments via `bin = floor(θ / (180/B))`, et comment les magnitudes sont additionnées dans chaque bin.

Nombre de compartiments (B) 2

📄 **Entrée (exactement comme le programme la lit via 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
```

🔢 **Matrice des magnitudes |∇f|**

1.0	2.0	0.5
3.0	4.0	1.5
0.8	2.5	3.2

📐 **Matrice des orientations θ (degrés) — colorée par bin**

10° ÷90.0=0.11 ⌊·⌋=bin 0	100° ÷90.0=1.11 ⌊·⌋=bin 1	45° ÷90.0=0.50 ⌊·⌋=bin 0
170° ÷90.0=1.89 ⌊·⌋=bin 1	20° ÷90.0=0.22 ⌊·⌋=bin 0	95° ÷90.0=1.06 ⌊·⌋=bin 1
60° ÷90.0=0.67 ⌊·⌋=bin 0	150° ÷90.0=1.67 ⌊·⌋=bin 1	5° ÷90.0=0.06 ⌊·⌋=bin 0

📏 **Où chaque θ tombe sur la règle [0°, 180°) — bin = floor(θ / largeur)**

(2,2)(0,0) (1,1) (0,2) (2,0) (1,2)(0,1) (2,1) (1,0)

0° 90° 180°

📊 **Plages de chaque compartiment (largeur = 180° / B)**

bin 0: [0.0°, 90.0°) bin 1: [90.0°, 180.0°)

🧩 **Chaque pixel : magnitude + orientation → bin**

mag=1.0 θ=10° → bin 0	mag=2.0 θ=100° → bin 1	mag=0.5 θ=45° → bin 0
mag=3.0 θ=170° → bin 1	mag=4.0 θ=20° → bin 0	mag=1.5 θ=95° → bin 1
mag=0.8 θ=60° → bin 0	mag=2.5 θ=150° → bin 1	mag=3.2 θ=5° → bin 0

Histogramme brut (H) | Histogramme normalisé (Ĥ)

9.50 bin 0 — 9.00 bin 1

🧮 **Calcul pas à pas (floor de la division + somme des magnitudes par 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.fr/cap07/sim-ep0705.html

Figure 8.25: Simulateur EP07_05 : Histogramme HOG d'une cellule (mapping des angles vers les bins)

7. **Matrice de confusion :** Construire une matrice $C \times C$ où la ligne correspond à la classe réelle et la colonne à la classe prédite, en suivant l'ordre des classes du point 1.
8. **Précision :** Calculer la précision globale comme le rapport entre les succès et Q.
9. **Sortie :** Pour chaque échantillon de test, dans l'ordre d'entrée, imprimer la classe prédite. Ensuite, imprimer la matrice de confusion (une ligne par classe réelle, valeurs séparées par des espaces, dans l'ordre des classes). Enfin, imprimer la précision arrondie à 4 décimales.

8.18.6.2 📌 Contraintes computationnelles

- **Métrique sélectionnable :** implémenter les deux distances ; la métrique M définit celle utilisée pour toute l'exécution (il n'est pas possible de mélanger les métriques dans le même appel).
- **Départage de vote déterministe :** le critère du point 6 (ordre de la liste des classes) doit être suivi même lorsque l'égalité implique plus de deux classes.
- **Indépendance de l'entraînement et du test :** il n'est pas nécessaire de vérifier que les échantillons de test n'apparaissent pas dans l'entraînement — supposer que l'entrée est valide.

8.18.6.3 🧠 Fondement théorique

Étape de l'exercice	Étape correspondante dans le chapitre
Histogrammes d'entraînement/test déjà extraits	`descritor_lbp` appliqué aux textures synthétiques
Distance euclidienne ou Manhattan	Paramètre `metric` du `KNeighborsClassifier`
Vote majoritaire avec k voisins	`KNeighborsClassifier.predict`
Matrice de confusion $C \times C$	`confusion_matrix` de `scikit-learn`
Précision globale	`accuracy_score` de `scikit-learn`

Cet exercice met en évidence, de manière contrôlée, un résultat discuté dans le chapitre : le **choix de la métrique de distance** et de la **valeur de** k peut modifier la classe prédite pour un même échantillon, même en maintenant fixe le descripteur utilisé — renforçant que, dans la reconnaissance de formes classique, le descripteur, la métrique et le classifieur forment un système interdépendant, et non des pièces isolées.

8.18.6.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : entier C suivi de C noms de classe.
- Ligne 2 : entier H, *string* M et entier k.
- Ligne 3 : entier N.
- Prochaines N lignes d'entraînement : nom de la classe suivi de H réels.
- Ligne suivante : entier Q.
- Prochaines Q lignes de test : nom de la classe réelle suivi de H réels.

Sortie :

- Q lignes avec la classe prédite de chaque échantillon de test, dans l'ordre d'entrée.
- C lignes avec la matrice de confusion (une ligne par classe réelle).
- Dernière ligne : `Acuracia: <valeur>`.

8.18.6.5 📌 Exemples

Entrée (résumée)	Sortie	Observation
2 granular listrada 2 euclidiana 1 4 granular 0.9 0.1 granular 0.8 0.2 listrada 0.1 0.9 listrada 0.2 0.8 2 granular 0.85 0.15 listrada 0.15 0.85	granular listrada 1 0 0 1 Acuracia: 1.0000	Avec $k = 1$, chaque test est classé par le voisin d'entraînement le plus proche.

Note

Ce simulateur utilise un ensemble simplifié de **3 classes** (`granulaire`, `striée`, `tachetée`) sur des points 2D fictifs, uniquement pour illustrer le *pipeline* de vote, de départage et de matrice de confusion du k-NN. Dans le cadre du **EP07_07**, vous appliquerez cette même logique à une mosaïque d'image réelle, qui introduit une quatrième classe (`échiquier`) et remplace les points 2D par des histogrammes LBP extraits directement des pixels de l'image.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap07/sim-ep0706.html

Figure 8.26: Simulateur EP07_06 : *Pipeline* k-NN Multi-Classe (vote, départage et matrice de confusion)

```
%%writefile EP07_06.py
# Code Python
```

```
Overwriting EP07_06.py
```

```
TestSuite("EP07_06.py").run()
```

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

8.18.7 EP07_07 ⚫ Classification réelle d'une mosaïque de textures via LBP + k-NN

Dans les exercices précédents, le descripteur LBP (**EP07_04**) et le classifieur k-NN multiclasse (**EP07_06**) ont été étudiés séparément, toujours à partir de données déjà fournies en entrée — voisinages 3×3 isolés ou histogrammes préalablement extraits. Dans cet exercice de clôture du chapitre, le programme devra **lire une image réelle**, au format **PGM ASCII (P2)**, calculer le descripteur LBP directement à partir des pixels, puis classer chaque région au moyen du k-NN, reproduisant ainsi, à échelle réduite, le flux complet d'un système de reconnaissance de textures. Cette approche anticipe également l'idée de **classification par mosaïque de régions**, liée à la segmentation sémantique étudiée dans un chapitre ultérieur.

Le simulateur interactif de l'**EP07_06** n'utilisait que trois classes (`granular`, `listrada` et `manchada`) représentées par des points bidimensionnels fictifs. Dans cet exercice, une quatrième classe, **xadrez**, est ajoutée, et les points sont remplacés par des histogrammes LBP extraits d'une image réelle.

L'image d'entrée est une **mosaïque** formée d'une grille $G \times G$ de blocs carrés de $S \times S$ pixels. Chaque bloc contient un échantillon de l'une des quatre classes de texture synthétique du chapitre : **granular**, **listrada**, **manchada** ou **xadrez** (motif en damier avec intensités alternées). Comme dans les autres exercices du livre, le chargement de l'image est réalisé par la fonction didactique `mm.readImg`.

> 💡 Pourquoi une mosaïque unique, et non plusieurs images ?
>
> L'entrée regroupe les $G \times G$ échantillons de texture dans un seul fichier **PGM**, uniquement pour simplifier la lecture des données et éviter l'ouverture de plusieurs fichiers. Pour l'algorithme, cela ne modifie pas le traitement : chaque bloc est traité de manière indépendante, comme s'il s'agissait d'une image isolée. La seule exception est l'**exclusion de la bordure** (point 4 ci-dessous).

8.18.7.1 📋 Directives d'implémentation

1. **Lecture des dimensions de l'image**

   Lire, via l'entrée standard, deux lignes contenant respectivement le nombre de lignes L et le nombre de colonnes C de la mosaïque (tous deux multiples de la taille de bloc S, avec $L = C$).

2. **Chargement de l'image**

   Utiliser la fonction didactique

   ```
   f = mm.readImg(L, C)
   ```

   pour lire les $L \times C$ valeurs d'intensité (niveaux de gris, `uint8`) de la mosaïque.

3. **Paramètres de la grille**

   Lire l'entier G (nombre de blocs par côté) et l'entier S (taille du côté de chaque bloc, en pixels), satisfaisant $L = C = G \times S$.

4. **Calcul du code LBP par pixel**

   Pour chaque pixel **intérieur** de l'image (c'est-à-dire ne se trouvant pas sur la bordure globale de `f` — ligne ou colonne 0 ou $L-1/C-1$), calculer le code LBP avec $P = 8$ voisins et rayon $R = 1$, en parcourant les voisins dans le sens **horaire** à partir du coin supérieur gauche, exactement comme dans l'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]`.

Les pixels sur la bordure globale de l'image **n'ont pas** de voisinage complet et doivent être **ignorés** (ils ne contribuent à aucun histogramme). Cela inclut les pixels de bordure qui tombent à l'intérieur d'un bloc (l'exclusion est toujours relative à la bordure de l'image entière, et non à celle de chaque bloc).

5. **Histogramme LBP uniforme par bloc (10 compartiments)**

   Pour chaque bloc (i, j) de la grille $(i, j = 0, \dots, G-1)$, accumuler, parmi ses pixels valides (point 4), un histogramme $H^{(i,j)}$ de 10 compartiments :

   - En considérant la séquence circulaire de bits $s_0, \dots, s_7$ du pixel (même règle de transitions que l'EP07_04) : si le nombre de transitions est ≤ 2 (motif **uniforme**), le pixel contribue au compartiment $\text{popcount}(s_0, \dots, s_7) \in \{0, \dots, 8\}$ (nombre de bits égaux à `1`) ;
   - Sinon (motif **non uniforme**), le pixel contribue au compartiment 9.

   À la fin, normaliser l'histogramme de chaque bloc en le divisant par le nombre de pixels valides qu'il contient, obtenant $\hat{H}^{(i,j)}$, avec $\sum_{b=0}^{9} \hat{H}^{(i,j)}[b] = 1$.

6. **Prototypes d'apprentissage**

   Lire l'entier Ncl (nombre de classes) suivi de Ncl noms de classes (ordre définissant la matrice de confusion et le départage des votes, comme dans l'EP07_06) ; ensuite, lire la *chaîne* M (métrique : `euclidiana` ou `manhattan`) et l'entier impair k ; enfin, lire l'entier N (nombre de prototypes) et, pour chacun, le nom de la classe suivi de 10 valeurs réelles (histogramme prototype déjà normalisé).

7. **Classification k-NN de chaque bloc**

   Pour chaque bloc, calculer la distance de $\hat{H}^{(i,j)}$ à chacun des N prototypes, en utilisant la métrique M (mêmes formules que l'EP07_06). Sélectionner les k prototypes les plus proches (départage des distances par l'ordre de lecture des prototypes) et classer selon la classe majoritaire (départage des votes par l'ordre des classes du point 6).

8. **Étiquettes réelles et évaluation**

   Lire, sur une seule ligne, les $G \times G$ noms de classes **réels** de chaque bloc, dans l'ordre de lecture par ligne de la grille (bloc $(0, 0)$, $(0, 1)$, …, $(0, G-1)$, $(1, 0)$, …). Construire la matrice de confusion $Ncl \times Ncl$ (ligne = classe réelle, colonne = classe prédite) et l'exactitude globale.

9. **Sortie**

   Imprimer, pour chaque bloc (dans le même ordre de lecture que les étiquettes réelles du point 8), la classe prédite. Ensuite, imprimer la matrice de confusion (une ligne par classe réelle, dans l'ordre du point 6). Enfin, imprimer l'exactitude, arrondie à 4 décimales.

8.18.7.2 📌 Contraintes informatiques

- **Descripteur fixe :** $P = 8$, $R = 1$ et 10 compartiments (selon le point 5) sont fixes dans cet exercice — ils ne sont pas lus depuis l'entrée.
- **Exclusion de bordure globale, non de bloc :** un pixel à la limite entre deux blocs, mais à l'intérieur de l'image, est valide et contribue normalement à l'histogramme du bloc auquel il appartient.
- **Ordre de lecture comme critère de départage :** tant le départage des distances (point 7) que celui des votes (point 7) suivent exactement les mêmes conventions que l'EP07_01 et l'EP07_06.
- **Prototypes en entrée, non appris :** contrairement au Projet Pratique 2, les histogrammes d'apprentissage sont fournis directement en entrée ; le programme ne doit pas générer de textures synthétiques.

8.18.7.3 🧠 Fondements théoriques

Étape de l'exercice	Étape correspondante dans le chapitre
Lecture de l'image via `mm.readImg`	Acquisition de l'image dans le *pipeline* de reconnaissance de formes
Code LBP par pixel (EP07_04)	`local_binary_pattern(image, P=8, R=1, method="uniform")`
Histogramme de 10 compartiments par bloc	Fonction `descritor_lbp` du Projet Pratique 2 (`bins=10`, `range=(0, P+2)`)
Classification k-NN avec métrique sélectionnable (EP07_06)	`KNeighborsClassifier` entraîné sur `X_textura`
Matrice de confusion $Ncl \times Ncl$ et exactitude	`confusion_matrix` et `accuracy_score` sur `yt_teste`

Cet exercice met en évidence, avec des pixels réels plutôt que des valeurs synthétiques, une limitation discutée dans la section finale du chapitre : des classes de texture visuellement distinctes pour un observateur humain — comme **granular** et **manchada** — peuvent produire des histogrammes LBP similaires lorsque le voisinage considéré est petit ($R = 1$), car toutes deux présentent une fréquence élevée de motifs non uniformes à l'échelle d'un seul pixel. La classe **xadrez**, quant à elle, possédant des bordures régulières et répétitives, tend à être séparée plus facilement. On s'attend à ce que la matrice de confusion produite reflète exactement ce schéma de confusion partielle.

8.18.7.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

```
L
C
[matrice L x C de l'image]
G S
Ncl nom_classe_1 ... nom_classe_Ncl
M k
N
nom_classe h0 h1 ... h9      (répétée N fois)
etiquette(0,0) etiquette(0,1) ... etiquette(G-1,G-1)
```

Sortie :

- $G \times G$ lignes avec la classe prédite de chaque bloc, dans l'ordre de lecture de la grille.
- Ncl lignes avec la matrice de confusion (une ligne par classe réelle, valeurs séparées par des espaces).
- Dernière ligne : `Acuracia: <valeur>`.

8.18.7.5 📌 Exemple (vérification manuelle)

Pour vérifier l'implémentation du descripteur avant de la tester sur une mosaïque complète, considérons une image 6×6 **homogène**, avec tous les pixels d'intensité 100, traitée comme un seul bloc ($G = 1$, $S = 6$). Comme tout pixel intérieur a ses 8 voisins d'intensité égale à celle du centre ($g_p \geq g_c$ dans tous les cas), tous les bits s_p valent `1`, le nombre de transitions est 0 (uniforme) et le compartiment est $\text{popcount}(11111111) = 8$. L'histogramme de l'unique bloc est donc `0 0 0 0 0 0 0 0 1 0`.

Entrée (résumée)	Sortie	Observation
6 6 [36 valeurs égales à 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	La distance du bloc au prototype `uniforme` est exactement 0 ; la classe `outra` n'apparaît pas dans l'étiquette réelle, c'est pourquoi sa ligne dans la matrice de confusion est nulle.

8.18.7.6 📌 Fichiers de référence (.pgm)

Pour le débogage local, deux mosaïques de test au format ASCII P2 sont fournies (annexées à cette livraison ; lors de leur intégration au référentiel du chapitre, enregistrez-les dans `all/cap07/dados/EP07/`) :

- 📥 **Cas 1 — Mosaïque simple (`Caso1_Mosaico_Simples.pgm`)** : grille 2×2 de blocs de 24×24 pixels, un échantillon de chacune des quatre classes, avec faible bruit — utile pour valider la lecture de l'image et la logique de classification dans un scénario contrôlé.
- 📥 **Cas 2 — Mosaïque mixte (`Caso2_Mosaico_Misto.pgm`)** : grille 3×3 de blocs de 16×16 pixels, avec classes répétées et plus grande variabilité — scénario dans lequel la confusion entre **granular** et **manchada** discutée dans les Fondements théoriques tend à se manifester.

La Figure 8.27 affiche les deux mosaïques, pour une inspection visuelle avant l'implémentation.

```
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)

    # Créer le répertoire local s'il n'existe pas
    if not os.path.exists(diretorio_local):
        os.makedirs(diretorio_local)

    # Si le fichier n'existe pas localement, le télécharger depuis le dépôt distant
    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"Téléchargement {nome_arquivo} depuis 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)

# Garantit le téléchargement et obtient le chemin correct
arq_caso1 = garantir_e_baixar_arquivo("Caso1_Mosaico_Simples.pgm")
arq_caso2 = garantir_e_baixar_arquivo("Caso2_Mosaico_Misto.pgm")

# Lit les matrices PGM
caso1 = ler_pgm_p2(arq_caso1)
caso2 = ler_pgm_p2(arq_caso2)

mm.show(
    [caso1, caso2],
    titles=[
        "Cas 1 : Mosaïque Simple\n(blocs 2x2, 1 échantillon/classe)",
        "Cas 2 : Mosaïque Mixte\n(blocs 3x3, classes répétées)",
    ],
    cols=2,
    figsize=(8, 4),
)
```

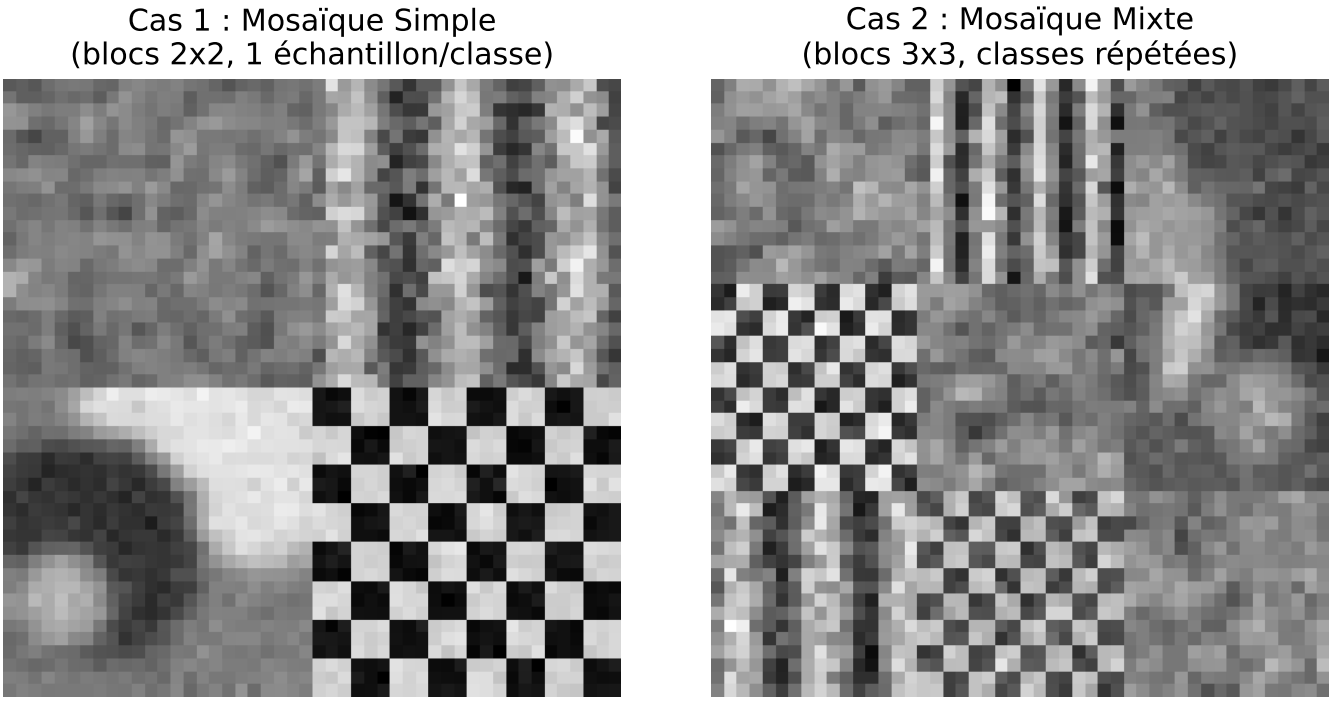


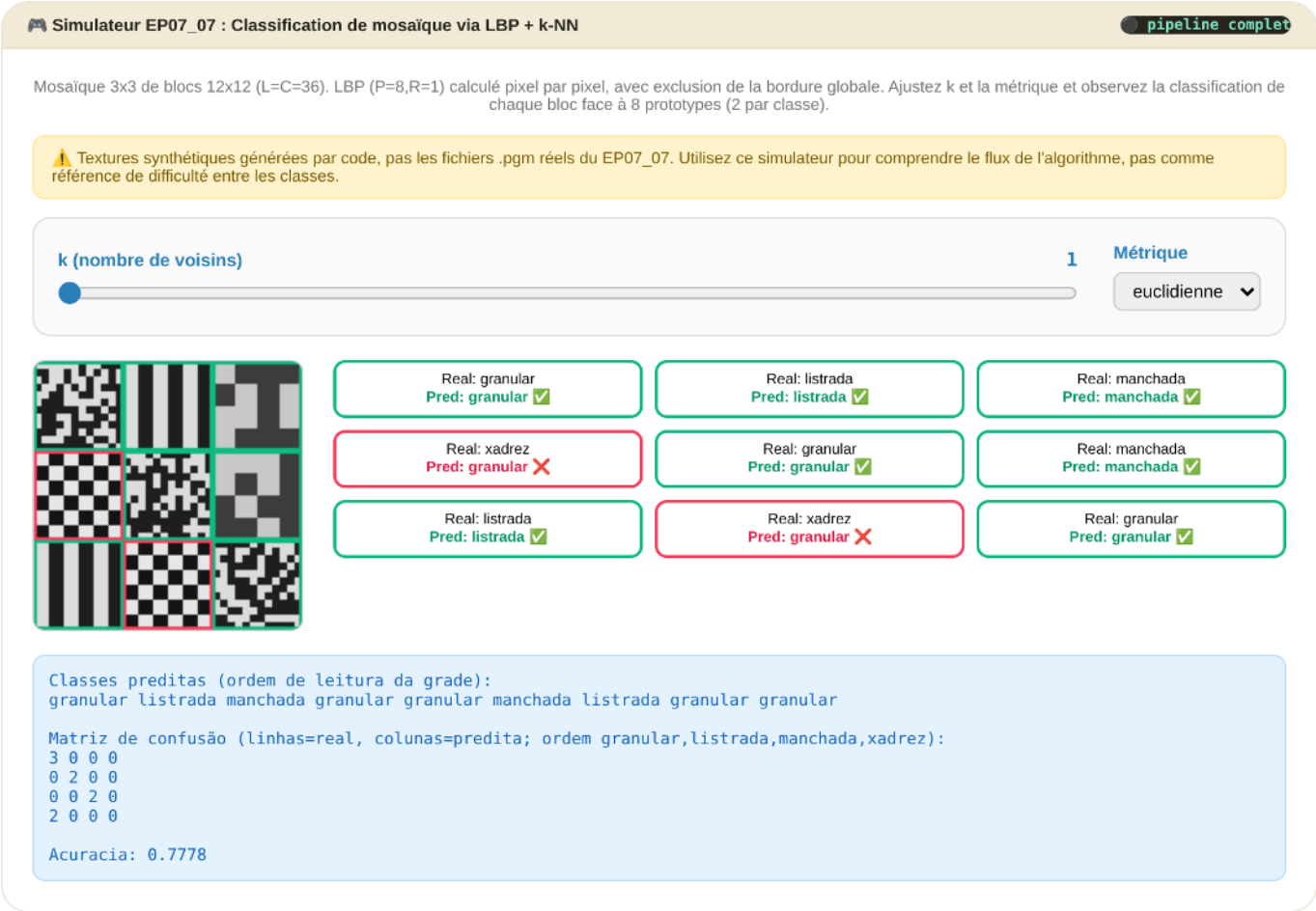
Figure 8.27: Mosaïques de référence (format PGM ASCII) utilisées dans l'EP07_07. Cas 1 : grille 2x2 avec un échantillon de chaque classe. Cas 2 : grille 3x3 avec classes répétées et plus grande variabilité.

```
%%writefile EP07_07.py
# Code Python
```

```
Overwriting EP07_07.py
```

```
TestSuite("EP07_07.py").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap07/sim-ep0707.html

Figure 8.28: Simulateur EP07_07 : Classification d'une mosaïque de textures via LBP + k-NN

Chapitre 9

Comprendre les scènes : Mise en correspondance de caractéristiques, détection et segmentation

Exécuter Colab Ouvrir GitHub

Dans le **chapitre 7**, nous avons étudié comment représenter des images à l'aide de descripteurs et utiliser ces représentations pour des tâches de classification. Dans ce chapitre, le problème est élargi : en plus de reconnaître des motifs, il devient nécessaire d'établir des correspondances entre différentes images, de localiser automatiquement des objets d'intérêt et d'interpréter l'organisation spatiale d'une scène.

Ces problèmes constituent certaines des principales tâches de la vision par ordinateur et représentent une étape naturelle après la classification d'images. Pour les résoudre, nous présenterons des méthodes classiques de mise en correspondance de caractéristiques, de détection d'objets et de segmentation d'images, ainsi qu'un aperçu des approches modernes basées sur l'apprentissage profond (*Deep Learning*), préparant la transition vers le **chapitre 9**.

9.1 Objectifs du chapitre

À l'issue de ce chapitre, l'étudiant devrait être capable de :

- **Établir des correspondances entre images** à l'aide de détecteurs et de descripteurs locaux, en estimant des transformations géométriques par le biais d'homographies pour le recalage et la correction de perspective ;
- **Localiser des objets d'intérêt** dans les images en utilisant des méthodes classiques de détection et évaluer les résultats à l'aide de métriques telles que l'***Intersection over Union*** **(IoU)** et de la technique de ***Non-Maximum Suppression*** **(NMS)** ;
- **Différencier les paradigmes de segmentation** — sémantique, par instances et panoptique — en comprenant que la segmentation panoptique unifie la segmentation sémantique et la segmentation par instances, offrant une description plus complète de la scène, et appliquer des méthodes classiques de segmentation ;
- **Extraire des descripteurs géométriques et topologiques** d'objets segmentés et les exporter sous forme d'annotations structurées (CSV ou format YOLO), en validant la qualité des annotations à l'aide de la métrique IoU ;
- **Relier les méthodes classiques de correspondance, de détection et de segmentation** aux approches modernes basées sur le *Deep Learning*, étudiées au **chapitre 9**.

La Figure 9.1 présente une vue d'ensemble des principaux concepts et des relations entre les sujets abordés dans ce chapitre, servant de carte conceptuelle pour guider la lecture.

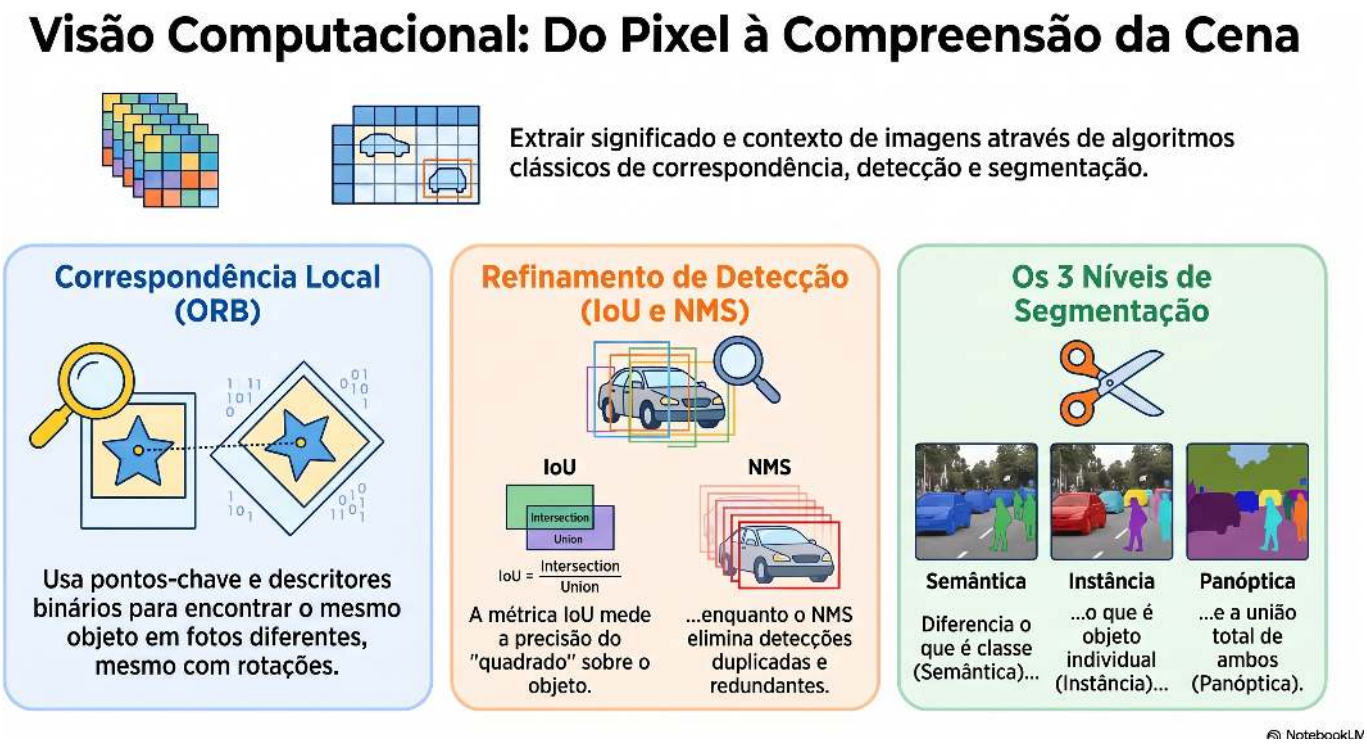


Figure 9.1: Vue d'ensemble des principaux concepts abordés dans ce chapitre, incluant la correspondance de caractéristiques, la détection d'objets, la segmentation d'images et leurs relations avec les approches modernes basées sur le *Deep Learning*. **Source :** élaboré avec l'aide du Gemini Notebook ({GOOGLE}, 2025).

9.2 Configuration de l'Environnement

```
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():
    """Installe les bibliothèques manquantes nécessaires pour ce chapitre."""
    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"[AVERTISSEMENT] Échec de l'installation de {pkg} (nécessaire pour le
                module {mod}).")


setup_cap08()

import cv2
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from skimage import data as skdata
from skimage.filters import threshold_otsu
from skimage.measure import label
from skimage.morphology import remove_small_objects, opening, disk
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0 | TestSuite : 1.1.2
```

9.3 Detecteurs et Descripteurs Locaux de Caractéristiques

Dans le chapitre précédent, chaque image était représentée par un unique vecteur de caractéristiques utilisé pour la classification. Dans diverses applications, cependant, il est nécessaire de comparer uniquement des parties de l'image, en établissant des correspondances entre des régions observées à différents instants, positions ou points de vue. Cette tâche requiert une représentation locale de l'image, capable d'identifier des structures suffisamment distinctes pour être retrouvées dans d'autres images.

Ce processus est réalisé en deux étapes complémentaires. Initialement, un **detecteur** identifie des points d'intérêt (*keypoints*), normalement associés à des coins ou à des régions présentant des variations significatives d'intensité. Ensuite, un **descripteur** représente numériquement le voisinage de chaque point détecté, permettant de comparer des régions correspondantes entre différentes images.

Une fois les paires *(point, descripteur)* obtenues, la mise en correspondance (*matching*) consiste à trouver, pour chaque descripteur d'une image, le descripteur le plus similaire dans l'autre. Ces correspondances constituent la base de diverses applications, telles que le recalage d'images, la reconstruction tridimensionnelle, la navigation visuelle et la réalité augmentée.

9.3.1 L'algorithme ORB (*Oriented FAST and Rotated BRIEF*)

Dans ce chapitre, nous utiliserons l'**ORB**, un détecteur et descripteur local qui combine efficacité computationnelle et robustesse aux rotations. L'algorithme réunit trois composantes principales :

- **FAST** (*Features from Accelerated Segment Test*), responsable de la détection des points clés ;
- **BRIEF** (*Binary Robust Independent Elementary Features*), responsable de la construction du descripteur binaire ;
- un mécanisme d'**estimation de l'orientation du voisinage**, qui rend le descripteur approximativement invariant à la rotation.

Le détecteur **FAST** parcourt tous les pixels de l'image. Pour chaque pixel candidat, il analyse un cercle de **16 pixels** autour de lui. Si un ensemble de pixels consécutifs présente une intensité significativement supérieure ou inférieure à celle du pixel central, ce pixel est considéré comme un **point clé** (*keypoint*). Ensuite, les candidats trop proches sont filtrés, ne conservant que les plus représentatifs.

Après la détection des points clés, le descripteur **BRIEF** est calculé dans un voisinage autour de chaque point clé, et non seulement sur les 16 pixels utilisés par FAST. Dans cette région, un **motif d'échantillonnage**, formé d'un ensemble fixe de paires de points (x, y) répartis dans une fenêtre autour du point clé, est utilisé pour effectuer des comparaisons d'intensité selon la Équation 9.1. Dans l'ORB, l'**orientation prédominante du**

voisinage est estimée à partir de la distribution des intensités dans cette région, et le motif d'échantillonnage est tourné en fonction de cette orientation. De plus, l'ORB utilise une version optimisée du BRIEF, appelée **rBRIEF** (*Rotated BRIEF*), dans laquelle les paires de points sont sélectionnées pour produire des descripteurs plus discriminatifs et avec une faible corrélation entre leurs bits.

$$\tau(p; x, y) = \begin{cases} 1, & \text{si } I(x) < I(y), \\ 0, & \text{sinon,} \end{cases} \tag{9.1}$$

où :

- p est le point clé détecté par FAST ;
- x et y sont deux points du voisinage de p, choisis par le motif d'échantillonnage du BRIEF ;
- $I(\cdot)$ représente l'intensité d'un pixel ;
- $\tau(p; x, y)$ est le résultat de la comparaison binaire entre les points x et y.

Chaque comparaison génère un bit du descripteur. La concaténation de toutes ces comparaisons forme le descripteur binaire associé au point clé.

Comme le descripteur est binaire, la similarité entre deux points est mesurée par la **distance de Hamming**, correspondant au nombre de bits différents entre deux descripteurs. Cette métrique peut être calculée de manière très efficace par des opérations logiques sur les bits, rendant l'ORB adapté aux applications en temps réel.

9.3.2 Exploration du simulateur ORB

La Figure 9.3 illustre, de manière interactive, la construction du descripteur **ORB**. Chaque segment représente l'un des **32 paires de points** (x, y) utilisés dans la Équation 9.1. La couleur du segment indique le résultat de la comparaison des intensités : **vert** lorsque $I(x) < I(y)$ (bit égal à 1) et **rouge** dans le cas contraire (bit égal à 0). La flèche **jaune** représente l'orientation prédominante du voisinage, estimée à partir du centroïde d'intensité. Le motif d'échantillonnage du BRIEF est pivoté en fonction de cette orientation, rendant le descripteur approximativement invariant à la rotation.

Dans un premier temps, laissez le bruit à zéro et comparez la séquence de bits avec le *curseur* à 0° puis à 15° (notez la valeur du champ « Signature du descripteur binaire » dans chaque cas) :

```
0°  : 00000000000000100000011111100010
15° : 00000000000000100000011111100010
```

Dans cet exemple, les deux descripteurs sont identiques, et le panneau « Dist. Hamming » confirme une distance nulle — preuve que l'estimation de l'orientation compense correctement la rotation de l'image.

Cliquez maintenant sur « Ajouter du bruit » et répétez l'expérience. Comme le bruit est généré de manière aléatoire à chaque exécution, les valeurs ci-dessous ne sont qu'un exemple — les vôtres seront différentes, mais doivent présenter une distance de Hamming d'amplitude similaire (généralement entre 6 et 14 bits, sur un total de 32) :

```
0°  : 01000000001000110110011111101010
15° : 10110000001011100100111111100010
```

Le bruit modifie une partie des comparaisons d'intensités, ce qui modifie certains bits du descripteur. La différence entre deux descripteurs est mesurée par la **distance de Hamming**, correspondant au nombre de positions où les bits diffèrent. Pour les descripteurs ci-dessus, cette distance est égale à **10**.

Une méthode efficace pour calculer cette distance en Python consiste à appliquer l'opération **XOR** (`^`), qui identifie les bits différents, suivie de la méthode `bit_count()`, qui compte le nombre de bits égaux à 1 dans le résultat.

```
def hamming(a: int, b: int) -> int:
    return (a ^ b).bit_count()
a = 0b01000000001000110110011111101010
b = 0b10110000001011100100111111100010
hamming(a, b)
```

```
10
```

Cette Figure 9.2 compare cette implémentation avec une version basée sur la comparaison de caractères et avec l'implémentation optimisée d'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   1.414191      1.0
1  XOR+bit_count()   0.013003    108.8
2           OpenCV   0.206146      6.9
```

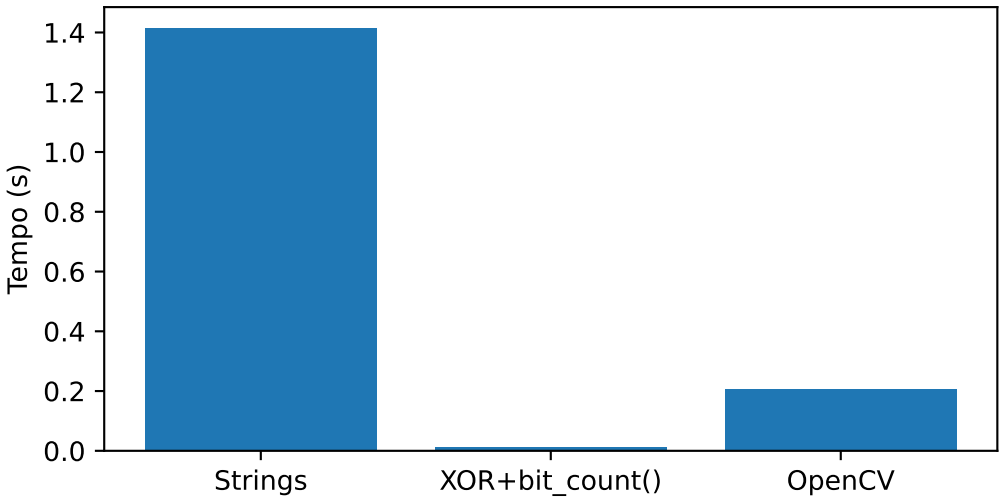


Figure 9.2: Comparaison des performances de différentes implémentations de la distance de Hamming.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap08/sim-08-orb.html

Figure 9.3: Simulateur interactif du descripteur ORB : explorez la logique de rotation et de construction du descripteur binaire. Modifiez la rotation pour observer comment le motif d'échantillonnage des tests binaires du BRIEF (lignes vertes et rouges) s'oriente dynamiquement pour garantir l'invariance angulaire.

9.4 Correspondance de caractéristiques et homographie avec ORB

Pour illustrer le processus de correspondance de caractéristiques, on utilise une scène synthétique obtenue par une rotation de 20° de l'image originale. Cette transformation simule une seconde capture de la même

scène sous un autre point de vue. La Figure 9.4 présente l'image de référence et sa version pivotée, qui seront utilisées dans les étapes suivantes.

```
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", "Rotation (20°)"],
        cols=2, figsize=(8, 4))
```

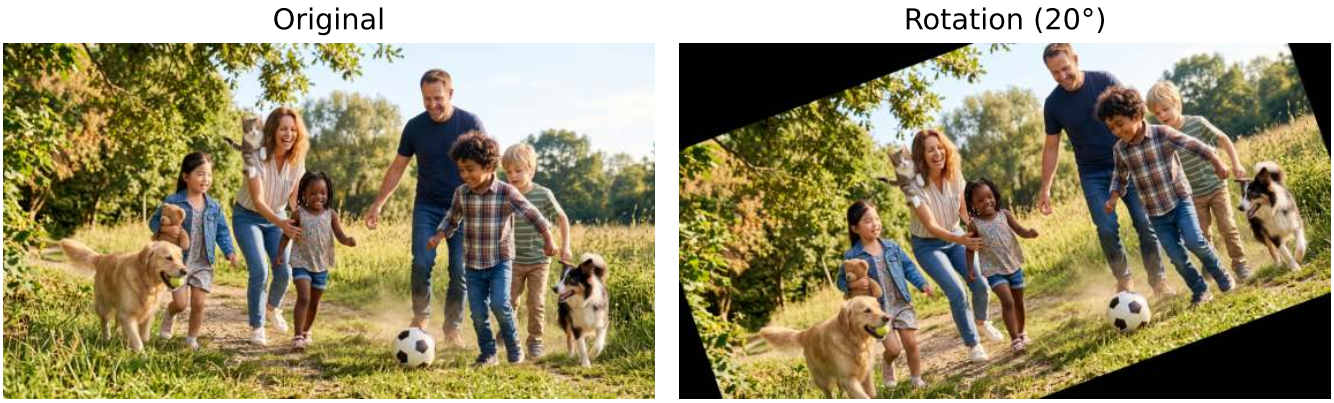


Figure 9.4: Image originale générée par Gemini et version pivotée (20°), simulant un changement de point de vue.

9.4.1 Détection et mise en correspondance des caractéristiques avec ORB

Avec les deux images disponibles, **ORB** détecte les points clés et calcule leurs descripteurs binaires. Ensuite, le **BFMatcher** de la bibliothèque `cv2` établit les correspondances entre les descripteurs en utilisant la distance de Hamming et la vérification mutuelle (*cross-check*). Enfin, les correspondances sont triées de la plus petite à la plus grande distance de Hamming, en privilégiant les paires potentiellement les plus fiables. Dans le code suivant, qui génère la Figure 9.5, les étapes suivantes sont mises en évidence :

1. `cv2.ORB_create(nfeatures=500)` instancie le détecteur ORB, limitant la recherche aux **500 points clés** les plus représentatifs de chaque image. Cette restriction réduit le coût computationnel et évite la sélection de points peu distinctifs.
2. `orb.detectAndCompute(...)` exécute, en un seul appel, la détection des points clés par **FAST** et le calcul des descripteurs par **BRIEF orienté**, renvoyant la liste des points clés (`kp`) et leurs descripteurs binaires de **256 bits** (`des`).
3. `cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)` crée un comparateur par force brute (*Brute-Force Matcher*), qui utilise la **distance de Hamming** — la même métrique exploitée dans la Figure 9.3 — pour comparer chaque descripteur de l'image originale avec tous les descripteurs de l'image pivotée. Le paramètre `crossCheck=True` ne conserve que les paires où la meilleure correspondance est réciproque, c'est-à-dire lorsque le meilleur correspondant de A est B et, simultanément, le meilleur correspondant de B est A. Ce critère élimine une grande partie des correspondances ambiguës.
4. `matches = sorted(...)` trie les correspondances de la plus petite à la plus grande distance de Hamming. Plus cette distance est petite, plus la similarité entre les descripteurs est grande et, par conséquent, plus la probabilité que la correspondance soit correcte est élevée.

La Figure 9.5 présente uniquement les cinq correspondances avec la plus petite distance de Hamming. Bien que ces paires soient les plus prometteuses, il n'existe encore aucune contrainte géométrique entre les points correspondants. En conséquence, certaines liaisons peuvent représenter de fausses correspondances (*false matches*), justifiant l'utilisation du **RANSAC** à l'étape suivante pour identifier uniquement les correspondances géométriquement cohérentes.

```
orb = cv2.ORB_create(nfeatures=500)
kp1, des1 = orb.detectAndCompute(img_original, None)
kp2, des2 = orb.detectAndCompute(img_cena, None)
print(f"Points d'intérêt détectés : {len(kp1)} (originale), {len(kp2)} (scène)")

bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = sorted(
    bf.match(des1, des2),
    key=lambda m: m.distance
)
print(f"Correspondances trouvées : {len(matches)}")

def draw_matches_destacado(img1, kp1, img2, kp2, matches, espessura=2, raio_ponto=4,
                           seed=42, cor_fixa=None):
    """Dessine les images côte à côte avec des lignes reliant les points correspondants.
    Si cor_fixe=None, chaque correspondance reçoit une couleur aléatoire (facilite la
distinction
    des liaisons individuelles). Si cor_fixe est définie (ex : vert), toutes les lignes
utilisent la
    même couleur - utile pour mettre en évidence un sous-ensemble spécifique, comme les
inliers du 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)  # graine fixe = couleurs reproductibles à chaque
exécution

    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

# essayez les 5 pires : 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 Correspondances ORB"], cols=1, figsize=(10, 5))
```

```
Points d'intérêt détectés : 500 (originale), 500 (scène)
Correspondances trouvées : 300
```

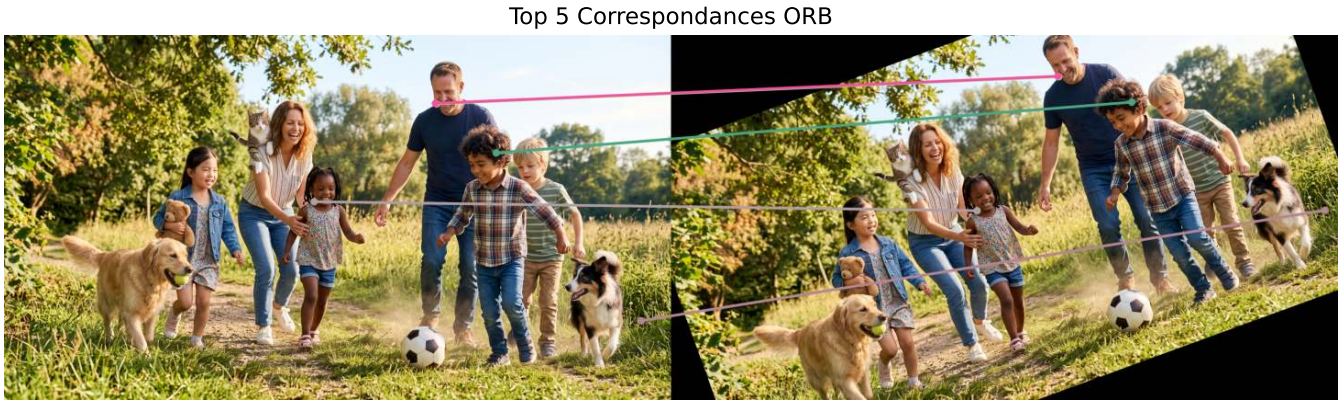


Figure 9.5: Les 5 meilleures correspondances de caractéristiques ORB entre l'image originale et la scène synthétique, avant le filtrage par RANSAC.

La fonction `draw_matches_destacado()` a uniquement une finalité de visualisation : elle positionne les images côte à côte et dessine des lignes entre les paires correspondantes, sans interférer avec l'estimation des correspondances.

9.5 Modélisation mathématique : homographie et RANSAC

9.5.1 Homographie

Dans les exercices 10 et 11 du chapitre 2, les fonctions `cv2.getPerspectiveTransform` et `cv2.warpPerspective` ont été utilisées pour corriger la perspective d'images à partir de **quatre paires de points correspondants** fournis manuellement. Dans ce chapitre, ces correspondances sont désormais obtenues automatiquement par ORB, ce qui permet d'estimer la transformation entre deux images sans intervention de l'utilisateur.

Mathématiquement, cette transformation est décrite par une **homographie**, représentée par une matrice 3×3 qui relie les coordonnées d'un même plan observé sous différents points de vue :

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

Après normalisation des coordonnées homogènes, on obtient le point correspondant

$$\left(\frac{x'}{w'}, \frac{y'}{w'} \right).$$

Comme l'homographie est définie à un facteur d'échelle près, elle possède **huit degrés de liberté**. Par conséquent, il faut au minimum **quatre paires de points correspondants** pour estimer ses paramètres.

En pratique, cependant, les correspondances produites automatiquement par ORB peuvent contenir des associations incorrectes (*outliers*). Pour estimer l'homographie de manière fiable même en présence de ces erreurs, on utilise l'algorithme **RANSAC**, présenté dans la section suivante.

9.5.2 RANSAC

Le **RANSAC** (*Random Sample Consensus*) est un algorithme d'estimation robuste capable d'ajuster un modèle géométrique même en présence d'observations incorrectes (*outliers*). Dans ce chapitre, le modèle d'intérêt est une **homographie**, estimée à partir des correspondances produites par l'ORB.

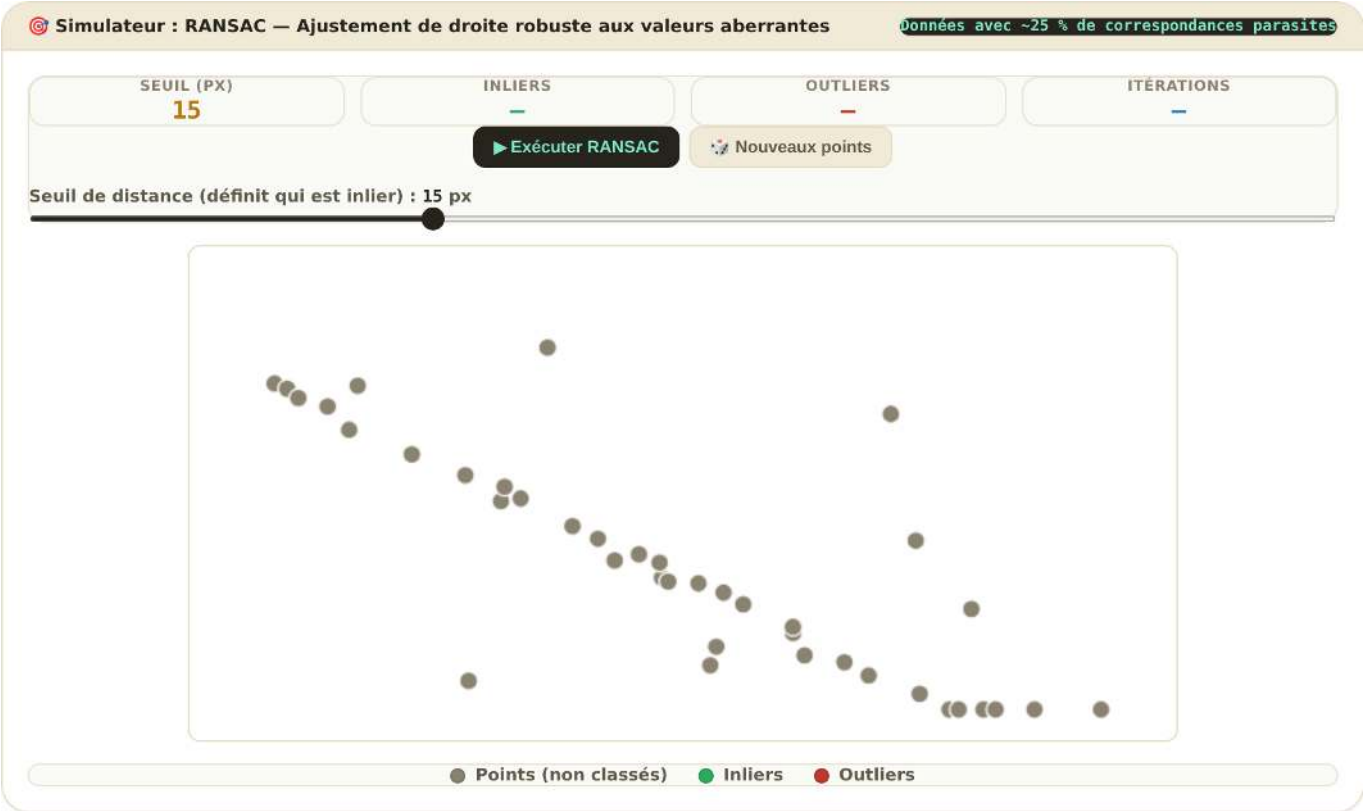
À chaque itération, l'algorithme :

1. sélectionne aléatoirement un petit sous-ensemble de correspondances (quatre paires de points, dans le cas de l'homographie) ;
2. estime une homographie candidate à partir de ce sous-ensemble ;
3. vérifie quelles correspondances sont compatibles avec cette transformation, en les classant comme ***inliers*** ou ***outliers*** ;
4. enregistre l'homographie qui produit le plus grand nombre d'***inliers*** ;
5. réestime l'homographie en utilisant uniquement les ***inliers*** trouvés.

Bien que l'exemple de ce chapitre utilise une homographie, le RANSAC est un algorithme à usage général et peut être employé pour estimer divers modèles géométriques, tels que des droites, des circonférences, des plans et d'autres transformations. Dans tous les cas, le principe est le même : générer des modèles candidats à partir de petits échantillons aléatoires et sélectionner celui qui présente le plus grand consensus parmi les données.

Pour comprendre ce processus de manière progressive, deux simulateurs sont présentés.

Le premier, illustré à la Figure 9.6, utilise l'exemple le plus simple possible : l'ajustement d'une **droite** à un ensemble de points contenant environ **25 % d'*outliers***. L'objectif est de comprendre les étapes fondamentales de l'algorithme — échantillonner, estimer un modèle, identifier les ***inliers*** et répéter le processus — sans la complexité du recalage entre images.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap08/sim-08-ransac.html

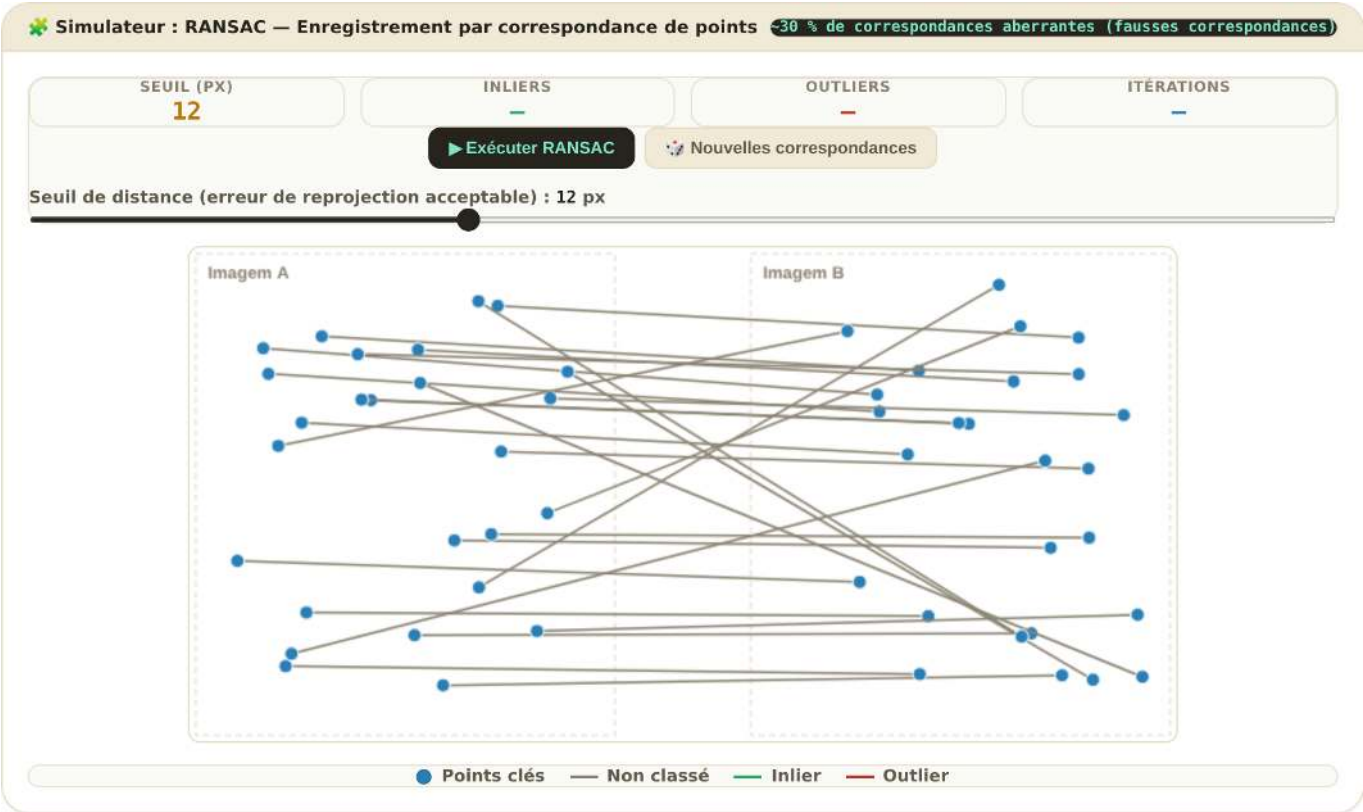
Figure 9.6: Simulateur interactif de l'algorithme RANSAC : ajustez le seuil de distance et exécutez l'algorithme pour observer la séparation entre *inliers* et *outliers*.

Le deuxième simulateur, présenté dans la Figure 9.7, se rapproche du problème étudié dans ce chapitre. Au lieu d'un seul ensemble de points, on considère **deux images** contenant des correspondances entre points clés. Certaines correspondances sont correctes (*inliers*), tandis que d'autres sont incorrectes (*outliers*), résultant d'erreurs dans le processus de mise en correspondance des descripteurs. Dans ce simulateur, le modèle estimé est une **transformation de similarité** (rotation, échelle et translation), plus simple qu'une homographie complète, mais suffisante pour illustrer le problème de recalage entre images.

Dans les deux simulateurs, l'algorithme exécuté suit exactement le même principe que celui utilisé ultérieure-

ment pour estimer l'homographie. La seule différence réside dans le modèle géométrique ajusté.

Ajustez le seuil de distance et exécutez l'algorithme dans chaque simulateur pour observer comment le RANSAC identifie les ***inliers***, rejette les ***outliers*** et estime un modèle cohérent en utilisant uniquement les correspondances valides.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap08/sim-08-ransac-registro.html

Figure 9.7: Simulateur interactif du RANSAC appliqué à l'enregistrement d'images : les points clés de deux images sont appariés par un descripteur, certaines correspondances sont erronées (*outliers*), et le RANSAC estime la transformation de similarité qui aligne la plupart d'entre elles.

9.5.3 Estimation de l'homographie avec RANSAC

Après avoir obtenu les correspondances entre les points clés par ORB, l'étape suivante consiste à estimer l'**homographie** entre les deux images. Pour cela, on utilise la fonction `cv2.findHomography()`, qui emploie l'algorithme **RANSAC** pour calculer cette transformation et renvoyer un masque indiquant quelles correspondances ont été classées comme ***inliers***.

Le code suivant réalise quatre opérations principales :

1. extrait les coordonnées des points correspondants dans chaque image ;
2. estime l'homographie à l'aide de `cv2.findHomography(..., cv2.RANSAC)` ;
3. reçoit le masque produit par RANSAC, dans lequel chaque correspondance est classée comme ***inlier*** ou ***outlier*** ;
4. utilise ce masque pour sélectionner uniquement les correspondances classées comme ***inliers***.

La Figure 9.8 présente uniquement les correspondances classées comme ***inliers***. On observe que ces paires de points sont compatibles avec une même transformation géométrique, tandis que les correspondances incohérentes (*outliers*) sont écartées. Par conséquent, l'homographie estimée représente de manière plus fidèle la relation géométrique entre les deux images.

```
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"Matrice d'homographie estimée :\n{H}\n")
print(f"Inliers : {n_inliers} sur {len(matches)} correspondances \
      ({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)  # vert (BGR)
)
mm.show([img_inliers], titles=[f"Correspondances Inliers (RANSAC) - \
                               {n_inliers}/{len(matches)}"], cols=1, figsize=(10, 5))
```

```
Matrice d'homographie estimée :
[[ 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 sur 300 correspondances       (95.0%)
```

Figure 9.8: Correspondances classées comme *inliers* (vert) par RANSAC lors de l'estimation de l'homographie entre les deux images.

9.5.4 Enregistrement de l'image

Après avoir estimé l'homographie, l'étape suivante consiste à l'utiliser pour enregistrer l'image de la scène dans le système de coordonnées de l'image originale. Ce processus permet d'aligner les deux images, facilitant ainsi leur comparaison.

Le code effectue trois opérations principales :

1. applique la transformation projective au moyen de `cv2.warpPerspective()`, en utilisant l'option `cv2.WARP_INVERSE_MAP`, qui applique en interne la transformation inverse sans qu'il soit nécessaire de calculer explicitement H^{-1} ;
2. calcule la différence absolue pixel par pixel entre l'image enregistrée et l'image originale au moyen de `cv2.absdiff()` ;
3. affiche l'image originale, la scène pivotée, l'image enregistrée et une carte des différences entre les deux images.

La Figure 9.9 présente le résultat de l'enregistrement. On observe que l'image enregistrée devient visuellement très proche de l'image originale, indiquant que l'homographie estimée a permis d'aligner correctement les deux vues de la même scène. La carte des différences met en évidence uniquement les régions où persistent

de petites discordances résultant d'erreurs d'interpolation, de quantification et de l'estimation elle-même de l'homographie.

```
h, w = img_original.shape[:2]

# WARP_INVERSE_MAP : applique H « de l'arrière vers l'avant », évitant le calcul manuel 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( # affiche mm.gray(erro)>10 en niveaux de gris
    [img_original, img_cena, img_registrada, mm.gray(erro)>10],
    titles=["Original", "Scène (rotationnée)", "Enregistrée", "Différence"],
    cols=2,
    figsize=(14, 8),
)
```

Figure 9.9: Enregistrement de l'image de la scène en utilisant l'homographie inverse estimée par RANSAC.

Pourquoi cela fonctionne-t-il ? — Robustesse par consensus

De nombreuses méthodes d'estimation ajustent un modèle en utilisant **toutes** les observations disponibles, en cherchant à minimiser l'erreur totale entre les données observées et le modèle ajusté (approche connue sous le nom de **moindres carrés**). Lorsqu'il existe des *valeurs aberrantes* (*outliers*), ces observations incorrectes peuvent décaler significativement le résultat obtenu.
Le **RANSAC** suit une stratégie différente. Au lieu d'utiliser toutes les données simultanément, il estime successivement des modèles à partir de petits échantillons aléatoires. Chaque modèle est ensuite évalué par le nombre de correspondances compatibles avec lui. À la fin des itérations, le modèle présentant le plus grand consensus parmi les données est sélectionné, c'est-à-dire le plus grand nombre d'***inliers*** (points concordants).
Deux paramètres jouent un rôle fondamental dans l'algorithme :

- le **seuil de reprojection**, qui définit la distance maximale pour qu'une correspondance soit classée comme ***inlier*** ;
- le **nombre d'itérations**, qui doit être suffisamment grand pour augmenter la probabilité de sélectionner au moins un échantillon exempt d'***outliers***.

Bien que très robuste, le RANSAC présuppose qu'il existe un modèle géométrique prédominant dans les données. Ses performances tendent à diminuer lorsque la proportion d'***inliers*** est très faible ou lorsque différentes structures géométriques coexistent dans la même scène, rendant difficile l'identification d'un unique modèle dominant.

9.6 Détection d'objets : *Haar Cascade* (Viola-Jones)

La correspondance de caractéristiques répond à la question : « **où se trouve le même objet ou motif observé précédemment ?** ». La **détection d'objets** résout un problème plus général : localiser automatiquement des instances d'une **catégorie** (par exemple, des visages humains), même si les objets spécifiques n'ont jamais été observés durant l'apprentissage. Alors que l'ORB nécessite deux images pour établir des correspondances entre points, le **Haar Cascade** opère sur une seule image, identifiant directement les régions candidates susceptibles de contenir l'objet recherché.

L'algorithme ***Haar Cascade***, proposé par Viola et Jones [Viola; Jones (2001); Viola (2004)], combine des caractéristiques de Haar, des images intégrales et une cascade de classifieurs pour réaliser la détection d'objets de manière efficace. Bien qu'il existe actuellement des méthodes plus récentes basées sur des réseaux neuronaux convolutifs, le Haar Cascade demeure disponible dans la bibliothèque OpenCV et constitue un exemple classique pour l'étude des techniques de détection d'objets.

Son fonctionnement repose sur trois composantes principales :

1. **Caractéristiques de Haar :** des filtres rectangulaires simples qui mesurent les différences d'intensité entre des régions voisines de l'image, exploitant des **motifs de contraste caractéristiques** de l'objet, comme la région des yeux généralement plus sombre que le front ;
2. **Image intégrale :** structure de données permettant de calculer rapidement la somme des pixels de toute région rectangulaire de l'image :

$$I_{\text{intégrale}}(x, y) = \sum_{x' \leq x,\, y' \leq y} I(x', y'),$$

   réduisant considérablement le coût computationnel de l'évaluation des caractéristiques de Haar. Avec cette structure, la somme des pixels de tout rectangle peut être obtenue avec seulement quatre accès à l'image intégrale ;
3. **Cascade de classifieurs :** durant l'apprentissage, l'algorithme AdaBoost sélectionne et combine des classifieurs simples en une séquence d'étapes. Lors de la détection, les régions qui **ne correspondent manifestement pas à l'objet** sont écartées dès les premières étapes, tandis que seules les candidates les plus prometteuses passent par les étapes suivantes, plus précises et computationnellement plus coûteuses. Cette stratégie permet d'effectuer une recherche efficace à différentes positions et échelles de l'image.

Le code suivant, qui génère la Figure 9.10, utilise des classifieurs préalablement entraînés et mis à disposition par OpenCV pour détecter des **visages**, puis restreint la recherche des **yeux** uniquement à l'intérieur de chaque visage détecté. Cette stratégie réduit les faux positifs et diminue le coût computationnel, car elle évite d'effectuer la recherche des yeux sur toute l'image.

Lors de la détection, une fenêtre parcourt l'image à différentes positions et échelles. La fonction `detectMultiScale()` effectue cette recherche automatiquement. Le paramètre `scaleFactor` contrôle le facteur de réduction entre les échelles consécutives de la fenêtre, tandis que `minNeighbors` définit le nombre minimal de détections voisines nécessaires pour confirmer un objet, réduisant ainsi les détections erronées. Le paramètre `minSize` établit la plus petite taille d'objet considérée durant la recherche.

La fonction retourne une liste de rectangles, chacun décrit par les coordonnées du coin supérieur gauche et les dimensions `(x, y, largeur, hauteur)`. Ces rectangles délimitent les régions classées comme objets par le détecteur et sont utilisés pour dessiner les boîtes montrées dans la Figure 9.10..

```
def get_cascade(nome):
    """Télécharge (si nécessaire) et charge un classificateur Haar Cascade d'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)

# Le plus utilisé
#face_cascade = get_cascade("haarcascade_frontalface_default.xml")

# Plus précis, mais plus lent
face_cascade = get_cascade("haarcascade_frontalface_alt2.xml")

# Compromis entre vitesse et précision
#face_cascade = get_cascade("haarcascade_frontalface_alt.xml")

# Très rapide, mais moins précis
#face_cascade = get_cascade("haarcascade_frontalface_alt_tree.xml")

# pour les yeux, le plus utilisé
#eye_cascade = get_cascade("haarcascade_eye.xml")
eye_cascade  = get_cascade("haarcascade_eye_tree_eyeglasses.xml")

img_original = mm.read(caminho_local)

# Échelle de gris + égalisation d'histogramme (Chapitre 3), comme attendu par Haar Cascade
img_rgb = mm.rotate(img_original, angle=0)
img_gray = cv2.equalizeHist(cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY))

# Détecte les visages et, dans chacun, tente de détecter les yeux
faces = face_cascade.detectMultiScale(
    img_gray,
    scaleFactor=1.1,      # Pas plus grands entre les échelles : plus rapide, mais moins
sensible
    minNeighbors=3,       # Nombre minimal de détections superposées pour confirmer un visage
    minSize=(90, 90)      # Ignore les régions candidates plus petites que 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"Régions détectées comme visage : {len(faces)}")
mm.show([img_anotada], titles=
        ["Détection Haar Cascade : visages (vert) et yeux (rouge)"],
        cols=1, figsize=(8, 8))
```

```
Régions détectées comme visage : 5
```

Détection Haar Cascade : visages (vert) et yeux (rouge)

Figure 9.10: Détection de visages et d'yeux avec *Haar Cascade*.

Pourquoi cela fonctionne-t-il ? — Et pourquoi cela échoue également

Dans l'image utilisée dans cet exemple, le classificateur détecte le visage et les yeux, mais il peut également marquer une seconde région sur une partie de l'arrière-plan de l'image comme s'il s'agissait d'un visage. C'est un exemple de **faux positif** : la distribution locale des intensités dans cette région est suffisamment similaire aux motifs appris lors de l'entraînement pour que la cascade la classe incorrectement comme un visage.

Ce comportement met en évidence l'une des principales limites du *Haar Cascade*. Comme la méthode fonde sa décision uniquement sur des caractéristiques de Haar, c'est-à-dire des différences d'intensité entre des régions rectangulaires, elle ne représente pas explicitement la forme ou la signification des objets présents dans l'image. Ainsi, des textures et des motifs de contraste similaires à ceux rencontrés dans les visages peuvent produire des détections incorrectes. De plus, ses performances tendent à diminuer face à de grandes variations de pose, des occlusions, des expressions faciales et des conditions d'éclairage différentes de celles présentes dans les données utilisées pour l'entraînement.

Malgré ces limites, le *Haar Cascade* reste utile dans les applications qui privilégient un faible coût computationnel. Dans les situations exigeant une plus grande capacité de généralisation face aux variations de l'apparence des objets, les méthodes modernes basées sur les réseaux neuronaux profonds présentent généralement de meilleures performances.

9.7 Détection d'Objets : Boîtes Englobantes, IoU et NMS

Bien que les méthodes de détection d'objets utilisent des stratégies assez différentes — du *Haar Cascade* aux détecteurs modernes basés sur des réseaux neuronaux profonds —, leurs résultats sont normalement représentés par des **boîtes englobantes** (*bounding boxes*), définies par les coordonnées $(x_{min}, y_{min}, x_{max}, y_{max})$.

9.7.1 *Intersection over Union* (IoU)

La métrique ***Intersection over Union*** **(IoU)** quantifie le chevauchement entre deux boîtes englobantes, par exemple, la détection produite par un algorithme et l'annotation de référence (*ground truth*) :

$$\mathrm{IoU}(A, B) = \frac{\mathrm{aire}(A \cap B)}{\mathrm{aire}(A \cup B)} \in [0, 1].$$

Plus la valeur est proche de 1, plus la concordance entre les boîtes est grande ; la valeur 0 indique une absence de chevauchement. Dans les évaluations des détecteurs d'objets, il est courant de considérer une détection comme correcte lorsque IoU $\geq 0{,}5$, bien que des applications spécifiques puissent adopter des seuils différents.

La métrique IoU n'est pas utilisée uniquement pour évaluer les détecteurs. Elle constitue également le critère employé par l'algorithme de **Suppression Non-Maximale** pour décider quand deux boîtes englobantes représentent le même objet et que, par conséquent, l'une d'elles doit être éliminée.

9.7.2 Suppression Non-Maximale (NMS)

Lors de la détection, il est courant que plusieurs boîtes englobantes soient associées au même objet. La **Suppression Non-Maximale** (*Non-Maximum Suppression*, NMS) élimine cette redondance en trois étapes :

1. trie les détections par score de confiance, de la plus élevée à la plus faible ;
2. conserve la boîte de plus haute confiance et élimine celles dont l'IoU avec elle dépasse un certain seuil ;
3. répète le processus avec les boîtes restantes jusqu'à ce qu'il n'existe plus de chevauchements pertinents.

L'exemple de la Figure 9.11 illustre cette procédure avec deux objets, chacun initialement représenté par plusieurs boîtes qui se chevauchent. Dans cet exemple, chaque boîte englobante reçoit un **score de confiance**, qui représente le degré de confiance du détecteur quant au fait que cette région contient l'objet recherché.

Au lieu d'implémenter l'algorithme manuellement, on utilise la fonction `cv2.dnn.NMSBoxes`, qui implémente la Suppression Non-Maximale employée dans divers détecteurs modernes. La fonction `desenhar_caixas` affiche, côte à côte, les boîtes avant et après le NMS, mettant en évidence la réduction de cinq détections à seulement deux, en préservant uniquement la boîte de plus haute confiance pour chaque objet.

```
def desenhar_caixas(caixas, pontuacoes, indices, cor, tamanho=(450, 450)):
    """Dessine les boîtes (et leurs scores) indiquées dans `indices` sur un fond blanc."""
    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 attend des boîtes au format (x, y, largeur, hauteur)
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,   # aucun seuil de confiance minimale ici
    nms_threshold=0.5      # seuil IoU pour considérer deux boîtes comme redondantes
).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"Avant le NMS ({len(caixas)} boîtes)", \
            f"Après le NMS ({len(mantidas)} boîte(s))"],
    cols=2, figsize=(9, 4.5)
)
```

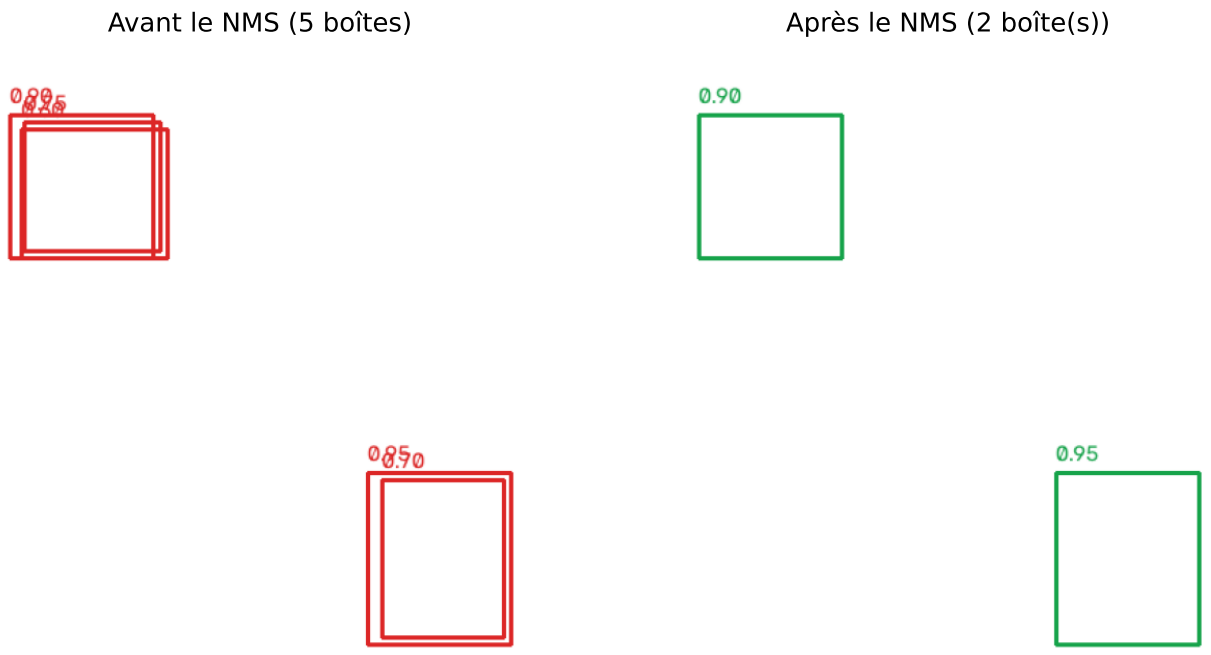


Figure 9.11: Effet de la Suppression des Non-Maximums (NMS) : les détections redondantes multiples (à gauche) sont réduites à une détection par objet (à droite).

Pourquoi cela fonctionne-t-il ? — Élimination des redondances

Le **NMS** (*Non-Maximum Suppression*) ne modifie pas la qualité des détections produites par le détecteur. Sa fonction est d'éliminer les **boîtes englobantes redondantes** qui représentent le même objet, en ne conservant que celle ayant le score de confiance le plus élevé.
Son efficacité dépend du seuil d'IoU adopté. Une valeur trop faible peut éliminer des boîtes correspondant à des objets différents dont le chevauchement est élevé, tandis qu'une valeur trop élevée peut maintenir plusieurs boîtes superposées pour un même objet.
Le NMS n'élimine pas non plus les **faux positifs isolés**. Si une région incorrecte est détectée par une seule boîte, elle sera conservée, car il n'existe pas d'autre détection redondante pour la comparer. Pour cette raison, le NMS est appliqué comme une étape de **post-traitement**, en utilisant uniquement les boîtes englobantes et leurs scores de confiance produits par le détecteur.

9.8 Segmentation visuelle : sémantique, par instances et panoptique

Une boîte englobante indique approximativement la position d'un objet, mais elle n'identifie pas quels pixels lui appartiennent. La **segmentation visuelle** résout ce problème en attribuant une étiquette à chaque pixel de l'image. Selon l'information produite, on distingue trois paradigmes principaux, résumés dans la Table 9.1.

Tableau 9.1: Comparaison des principaux paradigmes de segmentation visuelle.

Paradigme	Question à laquelle il répond	Distingue-t-il les objets d'une même classe ?
Sémantique	« À quelle classe appartient chaque pixel ? »	Non. Tous les pixels d'une même classe reçoivent la même étiquette.
Par instances	« Quels pixels appartiennent à chaque objet individuel ? »	Oui. Chaque objet reçoit un identifiant propre.
Panoptique	Combine les deux précédents.	Oui. Chaque pixel reçoit une classe et, lorsque cela s'applique, un identifiant d'instance.

Dans la segmentation **sémantique**, chaque pixel ne reçoit que l'étiquette de sa classe. Dans la segmentation **par instances**, en plus de la classe, les objets distincts appartenant à la même catégorie sont différenciés entre eux. Quant à la segmentation **panoptique**, elle combine ces deux informations, attribuant à chaque pixel une classe et, lorsque cela s'applique, un identifiant d'instance.

9.8.1 Segmentation sémantique par seuillage

Avant la popularisation des méthodes basées sur les réseaux de neurones profonds — thème du **Chapitre 9** —, de nombreuses applications de segmentation étaient résolues par des techniques classiques de traitement d'images. L'un des exemples les plus simples consiste à séparer l'objet et le fond par **seuillage**, produisant un masque binaire. Ce masque caractérise une **segmentation sémantique**, car chaque pixel appartient désormais à l'une des deux classes : **pièce** ou **fond**.

La Figure 9.12 utilise l'image de pièces de la bibliothèque `scikit-image`. Cet exemple reprend les concepts de seuillage et de morphologie mathématique étudiés au **Chapitre 4**, en remplacement de l'exemple précédent, qui utilisait une autre image de pièces.

Initialement, on applique une **fermeture morphologique** pour lisser les petites imperfections à la surface des pièces. Ensuite, le **seuillage d'Otsu** produit le masque binaire qui sépare les pièces et le fond. Après la suppression des objets connectés au bord de l'image, le remplissage des régions internes et l'élimination des petits bruits par opérations morphologiques, on obtient un masque sémantique adapté aux étapes suivantes.

9.8.2 Segmentation d'instances par étiquetage

Un masque sémantique ne fournit que la classe de chaque pixel, mais ne distingue pas les différents objets appartenant à la même catégorie. Pour obtenir une **segmentation d'instances**, on applique l'**étiquetage des composantes connexes**, qui attribue un identifiant différent à chaque région connexe du masque binaire.

Dans cet exemple, chaque composante connexe correspond à une pièce individuelle. Ainsi, l'étiquetage produit une segmentation d'instances, permettant d'identifier, de compter et de mesurer séparément chaque pièce présente dans l'image.

```
img_moedas = skdata.coins()

# 1. Fermeture morphologique : lisse les bords et comble les petites imperfections sur la
#    surface des pièces avant le seuillage
fechamento = mm.close(img_moedas, mm.sedisk(4))
```

```python
# 2. Seuillage d'Otsu (Chapitre 4) : sépare les pièces (claires) du fond (sombre)
mascara_bruta = mm.threshold(fechamento)

# 3. Supprime les objets connectés au bord de l'image - les pièces coupées aux
#    extrémités ne forment pas des instances complètes et perturberaient le comptage
mascara_sem_borda = mm.edgeoff(mascara_bruta)

# 4. Comblement des trous : remplit les éventuelles régions internes non détectées
#    par le seuil, garantissant que chaque pièce soit un disque solide
mascara_semantica = mm.clohole(mascara_sem_borda)

# 5. Ouverture morphologique : supprime le bruit résiduel et déconnecte les pièces qui
#    se touchent éventuellement, préparant le masque pour l'étiquetage
mascara_limpa = mm.open(mascara_semantica, mm.sedisk(6))

# 6. Étiquetage des composants connexes : chaque pièce isolée reçoit un
#    identifiant d'instance distinct
rotulos_instancias = mm.label(mascara_limpa)

print(f"Instances (pièces individuelles) identifiées : {np.max(rotulos_instancias)}")

mm.show(
    [img_moedas, mascara_bruta, mascara_sem_borda, mascara_semantica,
     mascara_limpa, rotulos_instancias],
    titles=[
        "Original",
        "1. Seuillage (Otsu)",
        "2. Suppression des bords",
        "3. Comblement des trous\n(masque sémantique : pièce x fond)",
        "4. Ouverture (nettoyage)",
        "5. Étiquetage\n(segmentation d'instances)"
    ],
    cols=3, figsize=(10, 6)
)
```

```
Instances (pièces individuelles) identifiées : 24
```

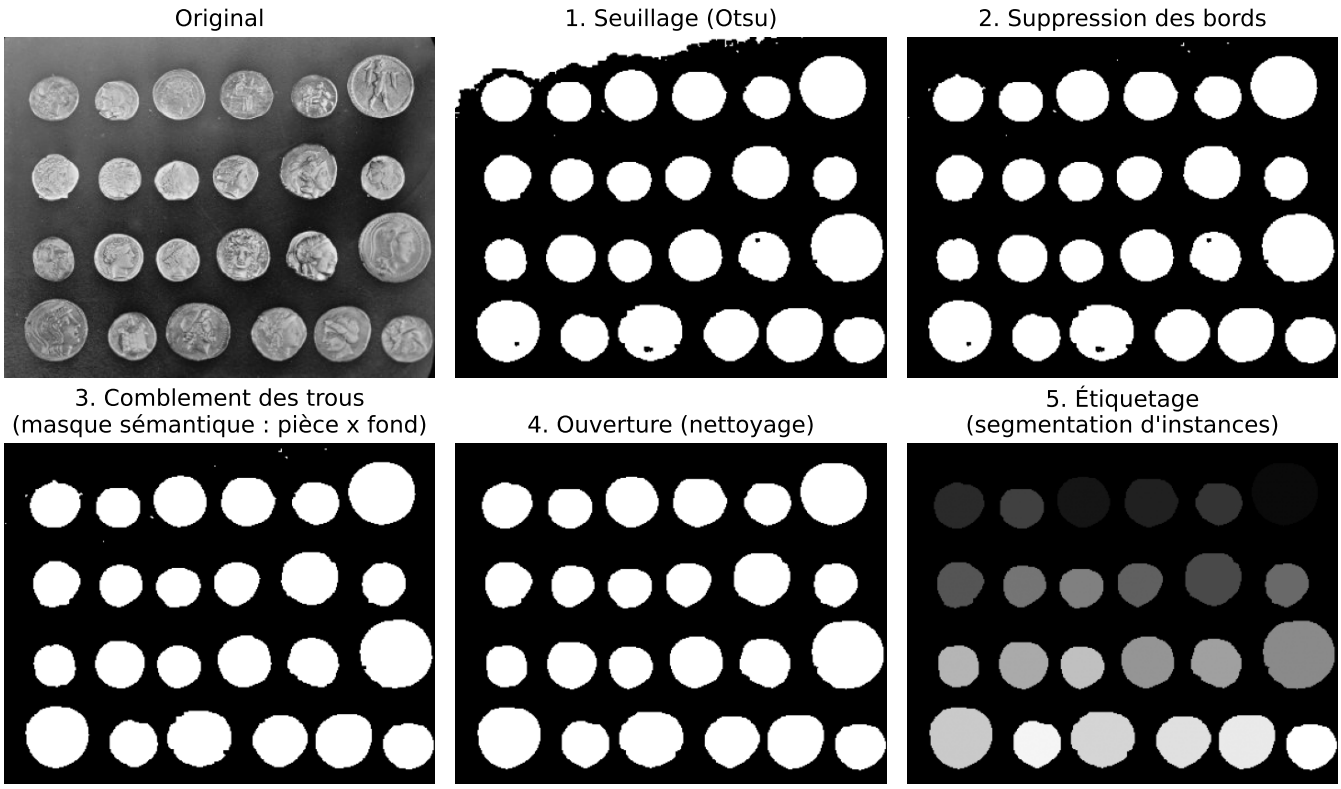


Figure 9.12: Segmentation classique (non basée sur l'apprentissage profond) : masque sémantique (pièce x fond) via seuillage d'Otsu, et segmentation d'instances via étiquetage des composants connexes.

> **Pourquoi cela fonctionne-t-il ? — Et où l'approche classique rencontre ses limites**
>
> Dans cet exemple, la segmentation est facilitée par le contraste entre les pièces et le fond, ainsi que par la relative homogénéité d'intensité à l'intérieur de chaque pièce. Le seuillage d'Otsu sépare efficacement ces deux régions, tandis que les opérations morphologiques éliminent les petites imperfections du masque binaire. Enfin, l'étiquetage des composantes connexes attribue un identifiant distinct à chaque région connexe, produisant ainsi une segmentation d'instances.
> Cette stratégie dépend toutefois directement de la qualité du masque binaire. Si deux objets sont joints, superposés ou présentent un contraste insuffisant par rapport au fond, ils peuvent être représentés par une unique région ou ne pas être segmentés correctement. De plus, les méthodes reposant principalement sur l'intensité des pixels ont une capacité limitée à distinguer des objets de classes différentes ayant une apparence similaire.
> Les méthodes modernes de segmentation fondées sur les réseaux de neurones profonds apprennent des représentations visuelles directement à partir des données d'entraînement, ce qui leur confère généralement une meilleure capacité à gérer les variations d'éclairage, de texture, de forme et d'occlusion. La segmentation panoptique élargit cette approche en combinant, dans une représentation unique, la classification sémantique de tous les pixels et l'identification individuelle des objets présents dans la scène.

9.8.3 Alternative : Transformée de distance + *Watershed*

L'ouverture morphologique peut séparer des pièces qui se touchent par le biais de l'érosion du masque binaire. Cependant, cette opération modifie le contour de tous les objets, y compris de ceux qui étaient déjà isolés. Une alternative, introduite au **Chapitre 4**, consiste à combiner la **transformée de distance** avec l'algorithme ***watershed***, en utilisant des marqueurs obtenus à partir du masque binaire lui-même.

La procédure se déroule en trois étapes :

1. on calcule la **transformée de distance** du masque binaire, en attribuant à chaque pixel de l'objet sa distance jusqu'au fond le plus proche ;
2. on applique un seuil à la transformée de distance pour obtenir des **marqueurs** situés dans les régions centrales des pièces ;
3. on utilise l'algorithme ***watershed*** pour étendre ces marqueurs jusqu'aux frontières entre les objets, séparant ainsi les pièces qui sont en contact.

La Figure 9.13 présente ces étapes. Les marqueurs sont obtenus dans les régions centrales de la transformée de distance et servent de graines pour le *watershed*, qui propage chaque étiquette jusqu'à rencontrer les frontières entre objets voisins.

En comparaison avec l'ouverture morphologique, les deux approches parviennent à séparer des pièces en contact. Cependant, comme le *watershed* utilise des marqueurs pour diviser les régions connectées, sans appliquer d'érosion directement sur le masque, les contours d'origine tendent à être mieux préservés, ce qui favorise l'obtention de mesures géométriques telles que l'aire, le périmètre et la circularité.

```
# Réutilise le masque sémantique AVANT l'ouverture agressive (sans érosion du contour)
mascara_base = mascara_semantica

# 1. Transformée de distance : chaque pixel du masque reçoit la distance jusqu'au fond proche
distancia = mm.dist(mascara_base)

# Marqueurs obtenus par seuillage de la transformée de distance.
# Les régions centrales des pièces restent connectées et sont utilisées
# comme graines pour l'algorithme watershed.
marcadores = mm.label(
    np.uint8(distancia > 0.5 * distancia.max())
)

# 3. Watershed : propage chaque marqueur dans le masque jusqu'aux points de contact entre
pièces
rotulos_watershed = mm.watershed(marcadores, mascara_base)

print(f"Ouverture morphologique (masque_propre) : {np.max(rotulos_instancias)} instances")
print(f"Distance + watershed : {np.max(rotulos_watershed)} instances")

mm.show(
    [mascara_base, distancia, mm.dil(marcadores,mm.sedisk(5)), rotulos_watershed],
    titles=[
        "Masque sémantique\n(sans ouverture, sans érosion)",
        "Transformée de distance",
        "Marqueurs\n(maxima régionaux)",
        "Watershed\n(instances séparées)"
    ],
    cols=4, figsize=(11, 3.2)
)
```

```
Ouverture morphologique (masque_propre) : 24 instances
Distance + watershed : 24 instances
```

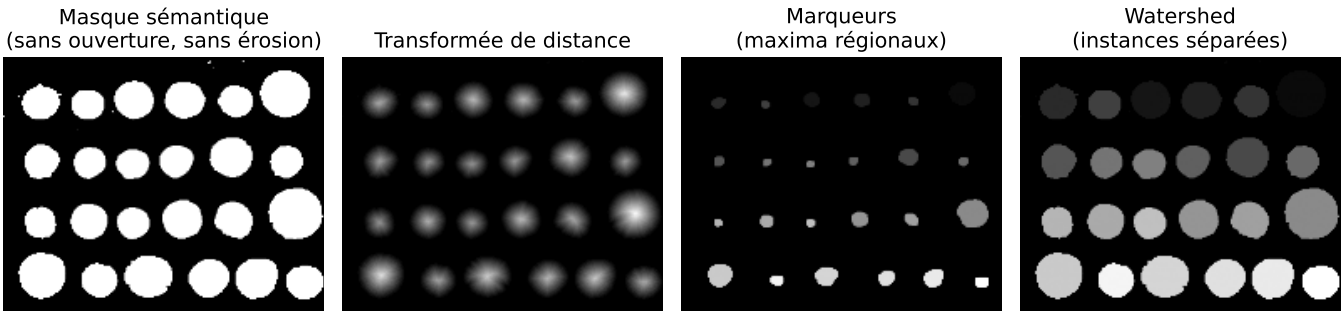


Figure 9.13: Séparation des pièces qui se touchent via transformée de distance + *watershed*, sans érosion du contour des pièces.

> 🧠 Pourquoi cela fonctionne-t-il ? — Marqueurs et *Watershed*
>
> La transformée de distance attribue des valeurs plus élevées aux pixels les plus éloignés du fond, qui se trouvent généralement dans les régions les plus centrales des objets. En appliquant un seuil à cette transformée, on obtient des marqueurs situés à l'intérieur de chaque pièce, favorisant l'obtention d'un marqueur pour chaque objet.
> L'algorithme ***watershed*** utilise ces marqueurs comme graines et propage leurs étiquettes jusqu'à ce que deux régions en croissance se rencontrent. Les points de rencontre entre ces régions définissent les frontières entre objets adjacents.
> La performance de cette méthode dépend de la qualité des marqueurs. Des objets très allongés, de formes irrégulières ou contenant de multiples maxima dans la transformée de distance peuvent générer des marqueurs supplémentaires, entraînant une sur-segmentation (*oversegmentation*). Dans des applications plus complexes, les méthodes modernes de segmentation d'instances basées sur des réseaux de neurones profonds apprennent directement, à partir des données d'entraînement, des représentations adaptées pour séparer les objets, évitant la construction explicite de marqueurs et d'autres heuristiques géométriques.

9.8.4 Mesure et Extraction des Attributs avec `mm.measure`

Après la segmentation et l'étiquetage des objets, l'étape suivante consiste à mesurer leurs propriétés géométriques. Pour cela, la bibliothèque `morph` met à disposition la fonction `mm.measure`, qui prend une image **binaire** et renvoie, pour chaque objet, un ensemble de descripteurs géométriques organisé dans une liste de dictionnaires.

Parmi les descripteurs calculés, on distingue l'**aire**, le **périmètre**, le **centroïde**, la **boîte englobante** (*bounding box*), la **circularité**, la **solidité** et le nombre de **sommets** du contour approximé. Ces sommets sont obtenus en approximant le contour par un polygone simplifié, calculé par la fonction `approxPolyDP`, qui préserve la forme générale de l'objet en utilisant un nombre réduit de segments.

La circularité est définie par

$$\text{Circularité} = \frac{4\pi \cdot \text{Aire}}{\text{Périmètre}^2},$$

prenant une valeur égale à 1 pour un cercle parfait et des valeurs inférieures pour des formes progressivement moins circulaires.

La solidité est donnée par

$$\text{Solidité} = \frac{\text{Aire}}{\text{Aire de l'Enveloppe Convexe}},$$

indiquant à quel point l'objet remplit son enveloppe convexe. Des valeurs proches de 1 caractérisent des objets convexes, tandis que des valeurs inférieures indiquent la présence de concavités.

Le nombre de sommets fournit une indication de la complexité de la forme de l'objet. Par exemple, un triangle tend à produire trois sommets et un rectangle quatre, tandis que les objets à contour courbe, comme les pièces de monnaie, aboutissent généralement à des polygones avec un plus grand nombre de sommets, selon la précision adoptée dans l'approximation.

Comme les pièces de monnaie de cet exemple présentent des contours approximativement circulaires et convexes, on s'attend à ce que la circularité prenne des valeurs élevées (typiquement entre 0,8 et 0,9) et que la solidité reste très proche de 1. Le nombre de sommets dépend du paramètre utilisé dans l'approximation polygonale (`precision`), augmentant à mesure que l'approximation préserve davantage de détails du contour. Ensemble, ces descripteurs peuvent être employés pour la classification d'objets, l'identification de composants parasites résiduels de la segmentation et la réalisation de mesures quantitatives sur l'image.

```
# mm.measure reçoit une image binaire ; chaque composant connexe correspond
# à une pièce individuelle.
medidas = mm.measure(mascara_limpa, precision=0.01)

if medidas:
    print(f"Nombre total d'objets catalogués : {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("Aucune mesure retournée.")
```

```
Nombre total d'objets catalogués : 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
```

On observe que **24 pièces** ont été identifiées. Les mesures de solidité restent proches de 1, indiquant des contours essentiellement convexes, tandis que la circularité varie entre environ 0,84 et 0,90 en raison des irrégularités du contour discrétisé. Le nombre de sommets varie entre 9 et 14, reflétant l'approximation polygonale utilisée pour représenter chaque pièce.

9.8.5 Exportation des annotations : de `mm.measure` au format YOLO

Outre les descripteurs géométriques, `mm.measure` calcule, pour chaque objet, sa **boîte englobante** (*bounding box*). Ces informations peuvent être exportées automatiquement par la fonction `mm.saveMeasures`, permettant de générer des fichiers d'annotation pour différentes applications sans nécessiter d'étiquetage manuel.

La fonction prend en charge trois formats de sortie :

- `fmt="csv"` : exporte tous les descripteurs géométriques, utile pour l'analyse exploratoire, la mesure et la classification basée sur des attributs ;
- `fmt="txt"` : enregistre les descripteurs dans un format tabulaire simple ;
- `fmt="yolo"` : génère des annotations compatibles avec le format utilisé par les détecteurs de la famille **YOLO** (*You Only Look Once*).

Dans le format YOLO, chaque objet est représenté par une ligne contenant

$$\text{classe} \;\; x_c \;\; y_c \;\; w_n \;\; h_n,$$

où (x_c, y_c) représente le centre de la boîte englobante et (w_n, h_n) ses dimensions, toutes normalisées par les dimensions de l'image :

$$x_c = \frac{x + w/2}{W}, \qquad y_c = \frac{y + h/2}{H}, \qquad w_n = \frac{w}{W}, \qquad h_n = \frac{h}{H},$$

où (x, y) correspondent aux coordonnées du coin supérieur gauche de la boîte englobante, (w, h) à ses dimensions en pixels et (W, H) à la largeur et à la hauteur de l'image. La normalisation rend les annotations indépendantes de la résolution de l'image, permettant d'utiliser le même format pour des images de tailles différentes.

Le code suivant exporte les mesures extraites des pièces de monnaie aux formats **CSV** et **YOLO**. La sortie CSV préserve tous les descripteurs géométriques calculés par `mm.measure`, tandis que le fichier au format YOLO ne contient que la classe et la boîte englobante normalisée de chaque objet, comme l'exigent les détecteurs de cette famille.

Cette représentation sera à nouveau utilisée dans le **Chapitre 9**, consacré aux méthodes de détection d'objets basées sur l'apprentissage profond (*Deep Learning*).

```python
# Exporte les mesures extraites des pièces en deux formats d'annotation
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("--- Extrait de moedas.csv ---")
with open("moedas.csv") as f:
    for linha in f.readlines()[:4]:
        print(linha.rstrip())

print("\n--- Extrait de moedas.txt (format YOLO) ---")
with open("moedas.txt") as f:
    for linha in f.readlines()[:4]:
        print(linha.rstrip())
```

```
--- Extrait 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
```

```
--- Extrait de moedas.txt (format 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
```

9.8.6 Validação de Anotações por IoU : `mm.verifyBoundBox`

Avant d'utiliser des annotations dans l'entraînement ou l'évaluation de modèles, il est important de vérifier qu'elles concordent avec un ensemble de référence (*ground truth*).

La fonction `mm.verifyBoundBox` effectue cette comparaison en utilisant la métrique **IoU** (*Intersection over Union*), présentée précédemment dans ce chapitre. Pour chaque boîte englobante candidate, la fonction calcule son chevauchement avec les boîtes de référence et comptabilise celles dont l'IoU est supérieure à un seuil spécifié.

Dans l'exemple suivant, on construit une **référence synthétique** à partir des cinq premières boîtes englobantes obtenues par `mm.measure`. Comme les annotations comparées sont les mêmes que celles utilisées pour construire la référence, on s'attend à ce que chaque objet trouve exactement une correspondance avec IoU $\geq 0{,}5$. L'objectif est uniquement d'illustrer le fonctionnement de la fonction `mm.verifyBoundBox` ; dans les applications réelles, la référence doit être obtenue indépendamment, par exemple par annotation manuelle.

```python
# Matrice de gabarits synthétique au format (classe, x1, y1, x2, y2), normalisée
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"Objet {m['id']}: {correspondencias} gabarit(s) correspondent avec IoU >= 0.5")

print(f"\nNombre total d'objets validés : {acertos}/{len(medidas[:5])}")
```

```
Objet 1: 1 gabarit(s) correspondent avec IoU >= 0.5
Objet 2: 1 gabarit(s) correspondent avec IoU >= 0.5
Objet 3: 1 gabarit(s) correspondent avec IoU >= 0.5
Objet 4: 1 gabarit(s) correspondent avec IoU >= 0.5
Objet 5: 1 gabarit(s) correspondent avec IoU >= 0.5

Nombre total d'objets validés : 5/5
```

On observe que les cinq objets ont été correctement associés à leurs gabarits respectifs, ce qui donne cinq correspondances valides. Dans des applications réelles, cette même stratégie peut être employée pour comparer automatiquement les boîtes englobantes produites par un détecteur avec des annotations de référence, permettant d'évaluer quantitativement ses performances au moyen de la métrique IoU.

9.8.7 Visualisation des annotations sauvegardées : `mm.showBoundBox`

Après l'exportation des annotations, il est utile de vérifier que les boîtes englobantes ont été correctement enregistrées. Pour cela, la bibliothèque `morph` met à disposition la fonction `mm.showBoundBox`, illustrée dans la Figure 9.14..

La fonction lit un fichier d'annotations aux formats `yolo`, `csv` ou `txt`, reconstruit les **boîtes englobantes** (*bounding boxes*) et les superpose à l'image originale, facilitant ainsi l'inspection visuelle du résultat. De cette manière, il devient possible de confirmer rapidement si les annotations sont alignées avec les objets, sans avoir besoin de les examiner manuellement.

Cette fonction complète `mm.verifyBoundBox`. Tandis que `mm.verifyBoundBox` effectue une **validation quantitative**, en comparant les annotations à un ensemble de référence à l'aide de la métrique IoU, `mm.showBoundBox` fournit une **validation qualitative**, permettant d'inspecter visuellement les boîtes englobantes reconstruites.

```
caminho_yolo = "moedas.txt"
img_anotada = mm.showBoundBox(img_moedas, caminho_yolo, fmt="yolo")
```

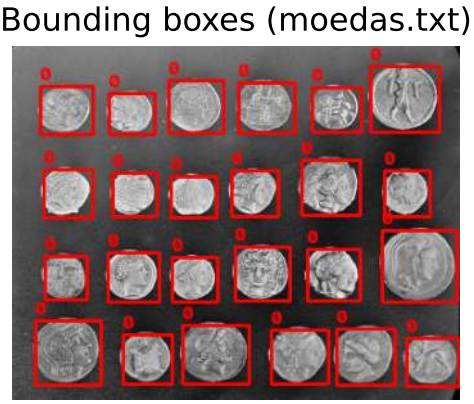


Figure 9.14: *Cadres englobants* reconstruits à partir du fichier d'annotations YOLO exporté par *mm.saveMeasures*, superposés à l'image originale des pièces.

> **Pourquoi cela fonctionne-t-il ? — De la segmentation aux annotations**
>
> Après avoir segmenté une image, il est possible de mesurer automatiquement chaque objet (`mm.measure`) et de convertir ces mesures en annotations (`mm.saveMeasures`) pour différents formats, y compris celui utilisé par les détecteurs de la famille YOLO. Cette procédure permet de générer automatiquement des ensembles initiaux d'annotations dans des scénarios contrôlés, réduisant considérablement le travail d'étiquetage manuel.
> Avant d'utiliser ces annotations lors de l'entraînement ou de l'évaluation de modèles, il est recommandé de les comparer à un ensemble de référence (*ground truth*). La fonction `mm.verifyBoundBox` automatise cette étape grâce à la métrique **IoU**, permettant de quantifier la concordance entre les boîtes englobantes générées et les annotations de référence. En complément, `mm.showBoundBox` permet une inspection visuelle des annotations reconstruites sur l'image originale, facilitant l'identification d'erreurs de normalisation, de positionnement ou d'ordonnancement des coordonnées qui peuvent ne pas être évidentes lors d'une validation purement numérique.
> Ensemble, `mm.measure`, `mm.saveMeasures`, `mm.verifyBoundBox` et `mm.showBoundBox` implémentent un flux complet pour mesurer des objets segmentés, générer des annotations, les évaluer et les inspecter visuellement. Lorsque la segmentation produit des résultats fiables, ce flux peut réduire considérablement le besoin d'étiquetage manuel dans la construction d'ensembles de données.

9.9 Aperçu des modèles modernes

Les techniques présentées dans ce chapitre — comme les détecteurs en cascade, la segmentation basée sur le seuillage et le post-traitement par IoU et NMS — restent importantes pour comprendre les fondements de la Vision par Ordinateur. Actuellement, cependant, de nombreuses applications utilisent des modèles d'**Apprentissage Profond**, capables d'apprendre automatiquement des représentations discriminatives à partir de grands ensembles de données, sans nécessiter la définition manuelle de caractéristiques.

La Table 9.2 présente quelques architectures représentatives pour la détection et la segmentation d'images. L'objectif est uniquement de les situer dans le contexte des tâches étudiées dans ce chapitre. Leurs principes de fonctionnement, leur entraînement et leur application seront discutés en détail dans le **Chapitre 9**.

Tableau 9.2: Architectures représentatives d'Apprentissage Profond pour la détection et la segmentation d'images.

Modèle	Tâche	Idée centrale
YOLO (*You Only Look Once*)	Détection d'objets	Détecte les objets en un seul passage à travers le réseau, en estimant les classes et les boîtes englobantes. Les versions récentes, comme le **YOLO26** (2026), éliminent l'étape de *Non-Maximum Suppression* (NMS), rendant l'inférence entièrement *end-to-end*.
Faster R-CNN	Détection d'objets	Génère des régions candidates et les affine avant la classification, en privilégiant la précision.
SSD (*Single Shot Detector*)	Détection d'objets	Détecte les objets à différentes échelles en un seul passage, cherchant un équilibre entre vitesse et précision.
U-Net	Segmentation sémantique	Produit un masque qui classe chaque pixel de l'image selon sa classe.
Mask R-CNN	Segmentation d'instances	Étend le Faster R-CNN en ajoutant un masque individuel pour chaque objet détecté.
***Segment Anything* (SAM)**	Segmentation orientée par *prompts*	Segmente les objets à partir d'indications telles que des points, des boîtes englobantes ou des masques. A évolué vers le **SAM 2**, avec prise en charge de la vidéo et du suivi temporel des objets, et le **SAM 3**, qui permet une segmentation guidée directement par des descriptions textuelles.

On observe une évolution des tâches abordées dans ce chapitre. Des modèles comme **YOLO**, **Faster R-CNN** et **SSD** localisent les objets à l'aide de boîtes englobantes. Le **Mask R-CNN** élargit cette capacité en produisant également un masque pour chaque instance détectée. Quant aux modèles plus récents, comme le ***Segment Anything* (SAM)**, ils permettent de segmenter des objets à partir de différents types de *prompts*, rendant le processus plus flexible.

Cette évolution accompagne la séquence de concepts développée tout au long du chapitre : de la correspondance de caractéristiques et de la localisation approximative par boîtes englobantes jusqu'à la segmentation précise des pixels appartenant à chaque objet. Dans le **Chapitre 9**, ces tâches seront revisitées sous la perspective de l'Apprentissage Profond, en explorant comment les réseaux de neurones modernes apprennent automatiquement des représentations capables de surmonter nombre des limitations des méthodes classiques présentées ici.

9.10 Limites des Approches Classiques et Motivation pour le *Deep Learning*

Les exemples développés dans ce chapitre illustrent trois problèmes fondamentaux de la vision par ordinateur : la **mise en correspondance de caractéristiques**, la **détection d'objets** et la **segmentation d'images**. Bien que les techniques classiques présentées soient efficaces dans divers scénarios, elles partagent toutes une limite importante : elles reposent sur des caractéristiques définies manuellement pour décrire ou identifier les objets d'intérêt.

- L'**ORB** utilise des descripteurs binaires capables d'établir des correspondances entre des points sous des rotations et des changements modérés d'échelle, mais il n'a pas été conçu pour reconnaître des catégories d'objets.
- Le **Haar Cascade** emploie un ensemble fixe de filtres rectangulaires, fonctionnant bien pour des objets à l'apparence relativement standardisée, comme les visages de face, mais devenant plus sujet aux faux positifs dans des scènes complexes.
- La **segmentation par seuillage** exploite les différences d'intensité entre l'objet et le fond, étant adaptée aux images à fort contraste, mais limitée face aux variations d'éclairage, aux textures ou aux scènes contenant plusieurs classes d'objets.

Dans tous ces cas, la performance dépend de la capacité des caractéristiques choisies à représenter adéquatement la variabilité présente dans les images. Dans les applications réelles, des facteurs tels que les changements de pose, l'éclairage, l'échelle, les occlusions et la diversité des catégories rendent cette tâche de plus en plus difficile, réduisant la généralisation des méthodes classiques.

Le **Chapitre 9** présente une approche différente pour ces problèmes au moyen des **réseaux de neurones convolutifs (CNN)**. Au lieu d'utiliser des caractéristiques définies manuellement, ces modèles apprennent automatiquement, à partir de grands ensembles de données, des représentations adaptées à chaque tâche. Les architectures modernes présentées dans la section précédente — comme **YOLO**, **Faster R-CNN**, **U-Net**, **Mask R-CNN** et ***Segment Anything*** **(SAM)** — suivent ce principe et représentent une évolution des techniques classiques étudiées dans ce chapitre, élargissant leur capacité à gérer la diversité et la complexité des images du monde réel.

9.11 Résumé

Dans ce chapitre, nous avons étudié les méthodes classiques de Vision par Ordinateur pour la **correspondance de caractéristiques**, la **détection d'objets** et la **segmentation d'images**. Enfin, quelques architectures modernes basées sur l'Apprentissage Profond ont été présentées, préparant la transition vers le **Chapitre 9**. Les principaux concepts abordés ont été :

- **ORB :** détection de points clés, construction de descripteurs binaires et correspondance de caractéristiques via la distance de Hamming.
- **Homographie et RANSAC :** estimation robuste de transformations projectives pour le recalage d'images et l'élimination de correspondances incohérentes.
- **Haar Cascade :** détection d'objets à l'aide de caractéristiques de Haar, d'image intégrale et de classificateurs en cascade.
- **Segmentation classique :** seuillage, opérations morphologiques, étiquetage de composantes connexes, transformée de distance et algorithme *watershed* pour la segmentation d'instances.

- **Mesure et annotation d'objets :** extraction d'attributs géométriques avec `mm.measure`, exportation automatique d'annotations (`mm.saveMeasures`) et validation par IoU (`mm.verifyBoundBox`) ainsi que par inspection visuelle (`mm.showBoundBox`).
- **Évaluation des détections :** utilisation de la métrique **Intersection sur Union (IoU)** et de la **Suppression des Non-Maxima (NMS)**.
- **Modèles modernes :** aperçu des architectures **YOLO**, **Faster R-CNN**, **SSD**, **U-Net**, **Mask R-CNN** et **Segment Anything (SAM)**, qui serviront de base à l'étude des méthodes d'Apprentissage Profond dans le **Chapitre 9**.

Étapes suivantes

Dans ce chapitre, des méthodes classiques ont été présentées pour résoudre les problèmes de **correspondance de caractéristiques**, de **détection d'objets** et de **segmentation d'images**, fondées sur des descripteurs, des filtres et des modèles géométriques définis manuellement.

Dans le **chapitre 9**, ces mêmes problèmes seront réexaminés sous l'angle de l'**apprentissage profond**, avec un accent mis sur les **réseaux de neurones convolutifs (CNN)**. Au lieu d'utiliser des caractéristiques conçues manuellement, ces modèles apprennent automatiquement des représentations à partir de grands ensembles de données, constituant ainsi la base des principaux systèmes modernes de vision par ordinateur pour la détection, la segmentation et la reconnaissance d'objets.

9.12 🤖 Utilisation du Gemini Notebook comme Aide à l'Étude

Le **Gemini Notebook** peut être utilisé comme outil complémentaire pour réviser les concepts présentés dans ce chapitre. À partir du contenu fourni comme référence, il permet de répondre à des questions, de rédiger des résumés, de clarifier des doutes et d'explorer les sujets de manière interactive.

Accès au Notebook du Chapitre

🚀 ACCÉDER AU GEMINI NOTEBOOK : CHAPITRE 08

🌐 **Langue et Langage de Programmation**

Le projet de ce chapitre dans le Gemini Notebook a été construit uniquement avec le texte en **portugais** et les exemples de code en **Python**. Si vous étudiez avec l'édition en anglais ou en français, ou que vous suivez le parcours en C++, les réponses du tuteur peuvent ne pas correspondre exactement à la version que vous lisez.
Les réponses générées par le Gemini Notebook sont produites automatiquement et peuvent contenir des imprécisions. Utilisez-les comme matériel complémentaire d'étude, en confrontant les informations avec le contenu de ce livre et, si nécessaire, avec d'autres sources académiques.

9.13 Liste d'exercices

Les exercices suivants consolident les concepts présentés dans ce chapitre au moyen d'adaptations, d'expériences et d'extensions des algorithmes développés tout au long du texte, en utilisant la bibliothèque didactique `morph`.

1. **(10 %)** Étudiez l'influence des rotations sur la stabilité du détecteur et du descripteur **ORB**. En utilisant l'image `skimage.data.astronaut()`, générez des versions tournées selon des angles de $0°, 45°, 90°, 135°, 180°$. Pour chaque cas, déterminez le nombre de correspondances obtenues par le `BFMatcher` et la fraction d'*inliers* identifiés par le **RANSAC**. Présentez les résultats dans un tableau et discutez de la robustesse de la méthode aux rotations analysées.

2. **(15 %)** Comparez le détecteur de coins **FAST** avec le détecteur de **Harris** (chapitre 6). Mesurez le temps moyen d'exécution et le nombre de points détectés dans les images `skimage.data.camera()`,

`skimage.data.gravel()` et `skimage.data.brick()`. Discutez des avantages et des limites de chaque détecteur en termes de vitesse et de répétabilité.

3. **(15 %)** Étudiez la sensibilité du **Haar Cascade** à la présence de bruit et de flou. Ajoutez un bruit gaussien ($\sigma \in 5, 10, 20$) et un flou de mouvement à l'image `skimage.data.astronaut()`. Faites varier les paramètres `scaleFactor` et `minNeighbors` de la fonction `detectMultiScale()` et discutez des effets sur les faux positifs, les faux négatifs et le nombre de visages détectés.

4. **(15 %)** Implémentez manuellement la métrique **Intersection over Union (IoU)** pour les boîtes englobantes et comparez vos résultats avec ceux de la fonction `mm.IoU`. Ensuite, créez une boîte de référence et dix boîtes obtenues par déplacements et variations d'échelle, en construisant un graphique reliant le déplacement appliqué à la valeur d'IoU obtenue.

5. **(15 %)** Évaluez l'algorithme de **Suppression Non Maximale (NMS)**. Créez une scène synthétique contenant trois objets, chacun représenté par au moins cinq boîtes se chevauchant avec différents scores de confiance. Exécutez l'algorithme pour des seuils d'IoU égaux à $0.2, 0.4, 0.6, 0.8$, visualisez les résultats et discutez de l'influence de ce paramètre sur le nombre de boîtes conservées.

6. **(15 %)** Appliquez le pipeline de segmentation présenté dans ce chapitre (seuillage d'Otsu, opérations morphologiques et `mm.measure`) à l'image `skimage.data.coins()`. Exportez les mesures en utilisant `mm.saveMeasures(fmt="csv")` et produisez des histogrammes des distributions d'aire, de circularité et de solidité. Discutez des attributs les plus appropriés pour caractériser les pièces de monnaie.

7. **(15 %)** Construisez une image synthétique contenant des cercles, des rectangles et des triangles, certains partiellement superposés. Réalisez la segmentation, extrayez les mesures avec `mm.measure` et générez automatiquement les annotations au format YOLO en utilisant `mm.saveMeasures(fmt="yolo")`. Ensuite, modifiez délibérément certaines *boîtes englobantes* et utilisez `mm.verifyBoundBox` pour évaluer, à différents seuils d'IoU, combien d'annotations restent valides. Discutez de l'influence du chevauchement entre les objets sur la qualité des annotations obtenues.

8. **(Bonus – 10 %)** Développez un classificateur simple, comme le **k-NN** (chapitre 7), en utilisant uniquement les attributs géométriques produits par `mm.measure` (aire, circularité, solidité et nombre de sommets). Utilisez un ensemble de formes synthétiques (cercles, rectangles et triangles) avec différentes échelles et rotations, évaluez la précision obtenue et discutez de la capacité de ces descripteurs à distinguer différentes classes d'objets.

Références du chapitre

Les concepts et algorithmes présentés dans ce chapitre s'appuient sur des références classiques et contemporaines de la littérature en PDI-VC :

- Gonzalez (2018), pour les fondements de la segmentation d'images, des opérations morphologiques, de l'étiquetage des composantes connexes et de l'extraction d'attributs géométriques.
- Szeliski (2022), pour les concepts de correspondance de caractéristiques, de transformations géométriques, d'homographies, de détection d'objets et de segmentation en vision par ordinateur.
- Rublee (2011), pour la description de l'algorithme **ORB** (*Oriented FAST and Rotated BRIEF*), utilisé pour la détection, la description et la correspondance de caractéristiques locales.
- Fischler (1981), pour la formulation originale de l'algorithme **RANSAC** (*Random Sample Consensus*), employé pour l'estimation robuste de modèles géométriques en présence de données aberrantes.
- Viola (2001), Viola (2004) et Lienhart (2002), pour les fondements du détecteur **Haar Cascade**, incluant les caractéristiques de Haar, l'image intégrale, les classifieurs en cascade et les extensions utilisées dans les implémentations modernes.
- Kirillov (2019), pour la définition de la **segmentation panoptique**, qui intègre la segmentation sémantique et la segmentation d'instances en une représentation unique.
- Redmon (2016), Ren (2015), Ronneberger (2015), He (2016) et Kirillov (2019), pour les modèles modernes basés sur l'apprentissage profond appliqués à la détection et à la segmentation d'images, incluant **YOLO**, **Faster R-CNN**, **U-Net**, **Mask R-CNN** et **Segment Anything (SAM)**.

Exécuter Colab | Ouvrir GitHub

9.14 💻 Partie Pratique avec Exercices de Programmation

La présente liste d'exercices de programmation (EP) consolide les formulations théoriques présentées tout au long du Chapitre 8 — Correspondance de Caractéristiques, Détection d'Objets et Segmentation Classique — à travers un parcours pratique appliqué. Comme dans le chapitre précédent, les EP isolent les **grandeurs intermédiaires** de chaque technique — la distance entre descripteurs binaires, les termes d'une image intégrale, le comptage des *inliers* d'un modèle candidat, le chevauchement entre boîtes englobantes et l'étiquette de chaque composant connecté — permettant de valider manuellement chaque étape du raisonnement sans dépendre d'OpenCV ni d'images externes.

L'enchaînement des exercices reproduit le flux conceptuel du chapitre : on commence par la **distance de Hamming**, cœur de la correspondance des descripteurs binaires tels que ORB ; on avance vers le comptage des ***inliers*** qui soutient le **RANSAC** dans l'estimation robuste d'une homographie ; on poursuit avec l'**image intégrale**, l'astuce computationnelle qui rend le ***Haar Cascade*** viable en temps réel ; on approfondit l'**IoU et la Suppression Non-Maximale**, le post-traitement commun à pratiquement tout détecteur d'objets ; et on conclut avec l'**étiquetage des composants connectés**, l'approche classique — et ses limites — pour segmenter des instances individuelles dans un masque binaire.

🎯 Objectif de ce Carnet

Le carnet permet de développer, valider, organiser et tester des solutions d'**Exercices de Programmation (EPs)** dans des environnements interactifs, comme Colab, avec les mêmes cas de test que Moodle, en les y copiant uniquement au moment d'enregistrer la note officielle.

Téléchargement

Téléchargez `morph.py` et `testsuite.py` en exécutant la cellule ci-dessous :

```
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
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0 | TestSuite : 1.1.2
```

Exécution des tests

Pour évaluer les tests, exécutez `TestSuite("EP08_01.extensão").run()` dans une nouvelle cellule, en remplaçant l'extension par celle du langage utilisé (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). Le système télécharge les cas de test depuis GitHub, exécute le programme et calcule automatiquement la note.

Pour tester directement du code Python, sans enregistrer de fichier, utilisez `run_code(codigo)` en passant le code comme *chaîne de caractères* dans une variable `codigo` :

```
codigo = """
# ... votre code ici ...
"""
```

```
TestSuite("EP08_01").run_code(codigo)
```

Résumé des méthodes de `morph.py` (Chap. 8)

La bibliothèque `morph.py` met à disposition des fonctions pour l'analyse des composants connexes, l'extraction des contours, les métriques géométriques et les annotations :

1. **Composants et contours (`connectedComponents`, `findContours`)** Étiquettent les régions connexes et extraient les contours des images binaires.
2. **Propriétés des contours (`contourArea`, `arcLength`, `convexHull`, `approxPolyDP`, `fitLine`)** Calculent l'aire, le périmètre, l'enveloppe convexe, l'approximation polygonale et l'ajustement de droite pour un contour.
3. **Géométrie englobante (`boundingRect`, `minAreaRect`, `boxPoints`, `minEnclosingCircle`, `fitEllipse`)** Déterminent les rectangles englobants (alignés ou orientés), les ellipses et le plus petit cercle circonscrit.
4. **Extraction et persistance des mesures (`measure`, `saveMeasures`)** Extraient les descripteurs géométriques des objets (aire, circularité, solidité, centroïde) et exportent les données vers CSV, texte ou format YOLO.
5. **Évaluation et visualisation (`IoU`, `verifyBoundBox`, `showBoundBox`)** Calculent l'intersection sur union (*Intersection over Union*), valident les boîtes englobantes avec des gabarits et dessinent les boîtes délimitantes annotées sur l'image.

9.14.1 EP08_01 🟢 Distance de Hamming et correspondance de descripteurs binaires

L'ORB, utilisé dans le Projet Pratique 1 de ce chapitre, décrit le voisinage de chaque point d'intérêt comme une séquence de bits — et, par conséquent, la comparaison entre deux descripteurs n'utilise pas la distance euclidienne du k-NN du Chapitre 7, mais plutôt la **distance de Hamming** : le nombre de positions où les bits diffèrent. Avant d'appeler `cv2.BFMatcher(cv2.NORM_HAMMING)`, vous avez été chargé d'implémenter manuellement cette correspondance (*matching*) par force brute — la même étape qui, exécutée en interne par OpenCV, précède l'estimation robuste de l'homographie par RANSAC.

9.14.1.1 📋 Directives d'implémentation

1. **Quantités :** Lire les entiers N et M — nombre de descripteurs extraits de l'image A et de l'image B, respectivement.
2. **Descripteurs de A :** Lire N lignes, chacune contenant un descripteur binaire (une *chaîne* de caractères `0` et `1`, tous de même longueur).
3. **Descripteurs de B :** Lire M lignes, dans le même format.
4. **Seuil :** Lire l'entier τ — distance de Hamming maximale acceptable pour considérer une correspondance valide.
5. **Distance de Hamming :** Pour deux descripteurs binaires a et b de même longueur,

$$d_H(a, b) = \sum_k \mathbb{1}[a_k \neq b_k],$$

   c'est-à-dire le nombre de positions où les bits diffèrent.
6. **Correspondance par voisin le plus proche :** Pour chaque descripteur a_i de A (i dans l'ordre de lecture, commençant à 0), calculer sa distance de Hamming à **tous** les descripteurs de B et trouver celui de distance minimale. En cas d'égalité entre deux ou plusieurs descripteurs de B avec la même distance minimale, choisir celui de **plus petit indice**.
7. **Filtrage par seuil :** Si la distance minimale trouvée est $\leq \tau$, la correspondance est valide ; sinon, a_i n'a pas de correspondance.
8. **Sortie :** Pour chaque i de 0 à $N-1$, dans l'ordre de lecture, imprimer une ligne : `i j d` s'il existe une correspondance valide (où j est l'indice du descripteur de B choisi et d sa distance), ou `i -1` sinon. À la fin, imprimer `Total correspondances valides : X`.

9.14.1.2 📌 Contraintes computationnelles

- **Même longueur :** tous les descripteurs (de A et de B) ont exactement le même nombre de bits.
- **Force brute :** comparer chaque descripteur de A à **tous** ceux de B — aucune indexation ni structure d'accélération n'est nécessaire.
- **Départage par plus petit indice en B**, et **jamais** par ordre de lecture de A (qui est déjà naturel, car chaque a_i est traité de manière indépendante).

9.14.1.3 🧠 Fondement théorique

Élément	Rôle dans la correspondance ORB
Descripteur binaire (BRIEF)	Chaque bit est le résultat d'une comparaison d'intensité entre deux pixels du voisinage
Distance de Hamming	Métrique de dissimilarité entre *chaînes* binaires ; beaucoup plus rapide à calculer que la distance euclidienne (opération XOR + comptage de bits)
Voisin le plus proche	Critère de correspondance : chaque point de A est apparié au point de B avec le descripteur le plus similaire
Seuil τ	Filtre les correspondances peu fiables avant même le RANSAC — mais, comme discuté dans le chapitre, certaines correspondances incorrectes passent encore, exigeant la robustesse du RANSAC

9.14.1.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entiers N et M.
- N lignes suivantes : un descripteur binaire par ligne (*chaîne* de `0` et de `1`).
- M lignes suivantes : un descripteur binaire par ligne, dans le même format.
- Dernière ligne : Entier τ.

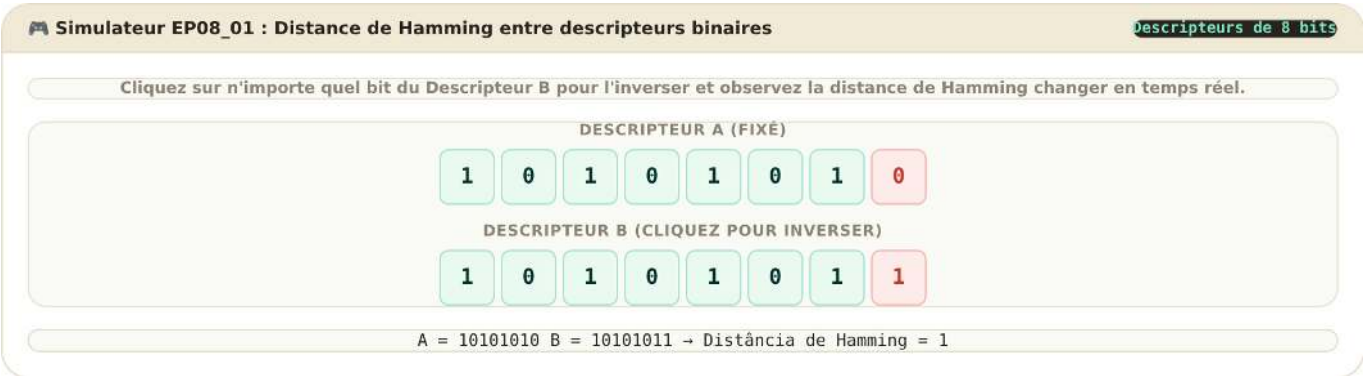
Sortie :

- N lignes, une par descripteur de A, au format `i j d` ou `i -1`.
- Dernière ligne : `Total correspondances valides : X`.

9.14.1.5 📌 Exemples

Entrée	Sortie	Observation
3 3 10101010 11110000 00001111 10101011 00001110 11111111 2	0 0 1 1 -1 2 1 1 Total correspondances valides : 2	Le descripteur `11110000` ne trouve pas de correspondance : son voisin le plus proche est à distance 4, au-dessus du seuil $\tau = 2$.

```
%%writefile EP08_01.py
# Code Python
```



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap08/sim-ep0801.html

Figure 9.15: Simulateur EP08_01 : Distance de Hamming entre deux descripteurs binaires

```
Overwriting EP08_01.py
```

```
TestSuite("EP08_01.py").run()
```

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

9.14.2 EP08_02 🟢 Homographie et RANSAC : Le vote par *inliers*

Le RANSAC, présenté dans la section « Modélisation mathématique : Homographie et RANSAC », répète un cycle de trois étapes — tirer un échantillon minimal, estimer un modèle candidat, et compter combien de correspondances sont cohérentes avec lui (les ***inliers***) — en conservant à la fin le modèle le plus voté. L'étape d'estimation du modèle à partir de 4 points (étape 2) implique une algèbre linéaire qui sort du cadre de cet EP ; ici, vous recevez directement un ensemble d'homographies **déjà candidates** — comme si chacune avait été estimée à partir d'un échantillon aléatoire différent — et vous êtes chargé de reproduire exactement l'étape décisive de l'algorithme : **appliquer chaque modèle à toutes les correspondances et compter ses *inliers***, en choisissant le gagnant.

9.14.2.1 📋 Directives d'implémentation

1. **Correspondances :** Lire l'entier N puis N lignes contenant chacune quatre réels, $x\ y\ x'\ y'$ — un point de l'image A et son correspondant (éventuellement incorrect) dans l'image B, exactement comme produit par l'étape de *matching* de l'EP08_01.
2. **Modèles candidats :** Lire l'entier K (nombre d'homographies candidates) et le réel ε (seuil d'erreur de reprojection). Ensuite, lire K lignes, chacune avec neuf réels $h_{11}\ h_{12}\ h_{13}\ h_{21}\ h_{22}\ h_{23}\ h_{31}\ h_{32}\ h_{33}$ — les éléments de la matrice H candidate, en ordre de lecture par ligne (*row-major*).
3. **Reprojection :** Pour chaque correspondance (x, y, x', y') et chaque modèle candidat H_k, calculer le point projeté
$$\begin{bmatrix}\hat{x}\\ \hat{y}\\ \hat{w}\end{bmatrix} = H_k \begin{bmatrix}x\\ y\\ 1\end{bmatrix}, \qquad (\hat{x}/\hat{w},\ \hat{y}/\hat{w}) \text{ est le point projeté.}$$
4. **Erreur de reprojection :** $e = \sqrt{(\hat{x}/\hat{w} - x')^2 + (\hat{y}/\hat{w} - y')^2}$.
5. **Comptage des *inliers* :** Une correspondance est un *inlier* du modèle H_k si $e \le \varepsilon$.
6. **Sélection du meilleur modèle :** Le modèle gagnant est celui ayant le plus grand nombre d'*inliers* ; en cas d'égalité, choisissez celui de **plus petit indice** k (le premier trouvé pendant le cycle itératif du RANSAC).
7. **Sortie :** Pour chaque modèle k de 0 à $K-1$, dans l'ordre de lecture, imprimer `Modelo k: I inliers`. À la fin, imprimer `Melhor modelo: k_best com I_best inliers`.

9.14.2.2 📌 Contraintes computationnelles

- **Comparaison inclusive :** une erreur de reprojection **exactement égale** à ε compte comme *inlier* ($e \leq \varepsilon$).
- **Sans estimation de H :** les matrices sont déjà fournies prêtes — il n'est pas nécessaire (ni attendu) de résoudre un quelconque système linéaire.
- **Égalité résolue par le plus petit indice**, reflétant le comportement naturel d'un algorithme itératif qui parcourt les modèles dans l'ordre et ne remplace le meilleur trouvé jusqu'alors que lorsqu'un nouveau modèle le **dépasse strictement**.

9.14.2.3 🧠 Fondement théorique

Élément	Rôle dans le RANSAC
Échantillon minimal (4 paires)	Suffisant pour déterminer les 8 degrés de liberté d'une homographie
Modèle candidat H_k	Estimé à partir d'un échantillon minimal ; peut être bon ou mauvais, selon que l'échantillon contenait des *outliers*
Erreur de reprojection	Mesure à quel point le modèle « prédit » chaque correspondance observée
Inlier vs. *outlier*	Correspondances cohérentes avec le modèle gagnant (*inliers*) vs. les autres, typiquement des correspondances incorrectes du *matching*
Raffinement final	En pratique, après avoir choisi le meilleur modèle, le RANSAC le recalcule en utilisant **uniquement** ses *inliers* — étape non exigée dans cet EP

9.14.2.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entier N.
- N lignes suivantes : quatre réels $x\ y\ x'\ y'$.
- Ligne suivante : Entier K et réel ε.
- K lignes suivantes : neuf réels (éléments de H_k, *row-major*).

Sortie :

- K lignes au format `Modelo k: I inliers`.
- Dernière ligne : `Melhor modelo: k_best com I_best inliers`.

9.14.2.5 📌 Exemples

Entrée	Sortie	Observation
5 0 0 0 0 1 1 2 2 2 0 4 0 0 2 0 4 5 5 1 1 2 0.5 2 0 0 0 2 0 0 0 1 1 0 0 0 1 0 0 0 1	Modelo 0: 4 inliers Modelo 1: 1 inliers Melhor modelo: 0 com 4 inliers	Le Modèle 0 (échelle $\times 2$) explique correctement 4 des 5 correspondances ; la 5 , $(5,5) \to (1,1)$, est un *outlier* qu'aucun des deux modèles n'explique bien.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap08/sim-ep0802.html

Figure 9.16: Simulateur EP08_02: RANSAC — Vote par Inliers entre Modèles Candidats

```
%%writefile EP08_02.py
# Code Python
```

```
Overwriting EP08_02.py
```

```
TestSuite("EP08_02.py").run()
```

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

9.14.3 EP08_03 🟢 Image intégrale : sommes rectangulaires en temps constant

Imaginez une caméra de sécurité traitant 30 images par seconde, et pour chaque image, le système doit balayer l'image à des dizaines de positions et d'échelles différentes, testant à chaque fois un ensemble de caractéristiques rectangulaires pour décider « y a-t-il un visage ici ? ». Si le calcul de la somme des intensités de chaque rectangle exigeait de sommer pixel par pixel, le système n'aurait aucune chance de fonctionner en temps réel — le goulot d'étranglement se situerait précisément dans la partie la plus répétée de l'algorithme. C'est exactement ce goulot d'étranglement que l'image intégrale élimine.

Le détecteur Haar Cascade évalue des milliers de caractéristiques rectangulaires par fenêtre, à de multiples positions et échelles — une approche irréalisable en temps réel si chaque rectangle exigeait la somme de ses pixels un par un. L'**image intégrale**, définie dans la section consacrée au Haar Cascade, résout ce problème : une fois précalculée, la somme des intensités de **n'importe quelle** région rectangulaire s'obtient avec seulement quatre consultations et trois opérations arithmétiques, quelle que soit la taille du rectangle.

Vous êtes chargé d'implémenter cette structure à partir de zéro : d'abord, calculer l'image intégrale à partir de l'image originale ; ensuite, répondre aux requêtes rectangulaires arbitraires.

9.14.3.1 📋 Directives d'implémentation

1. **Entrée :** Lire les dimensions $H \times W$ de l'image et ses $H \times W$ valeurs entières d'intensité.
2. **Image intégrale :** Calculer, pour chaque position (i, j) (indexation à partir de 0, `[ligne][colonne]`),
$$II(i,j) = \sum_{i' \le i,\ j' \le j} I(i', j'),$$
c'est-à-dire la somme de tous les pixels au-dessus et à gauche de (i, j), y compris la position elle-même.
3. **Requêtes :** Lire l'entier Q, puis Q lignes, chacune avec quatre entiers x_1 y_1 x_2 y_2 — les coins supérieur-gauche et inférieur-droit d'un rectangle, **tous deux inclusifs**, avec $0 \le x_1 \le x_2 < W$ et $0 \le y_1 \le y_2 < H$.
4. **Somme rectangulaire en O(1) :** Pour chaque requête, calculer la somme des intensités dans le rectangle en utilisant uniquement des valeurs déjà présentes dans II (sans parcourir les pixels originaux) :
$$S(x_1, y_1, x_2, y_2) = II(y_2, x_2) - II(y_2, x_1{-}1) - II(y_1{-}1, x_2) + II(y_1{-}1, x_1{-}1),$$

en traitant tout terme dont l'indice de ligne ou de colonne est égal à -1 comme 0.

5. **Sortie :** D'abord, imprimer l'image intégrale complète — H lignes avec W entiers chacune. Ensuite, pour chaque requête, imprimer un seul entier : la somme de la région correspondante.

9.14.3.2 📌 Contraintes de calcul

- **Ne recalculez pas par force brute :** la réponse à chaque requête doit utiliser la formule à quatre termes sur II, et non une somme directe des pixels du rectangle (même si le résultat numérique est le même, l'objectif de l'exercice est précisément cette technique).
- **Rectangles à coordonnées inclusives :** (x_1, y_1) et (x_2, y_2) appartiennent à la région sommée.
- **Traitement des bords :** lors de la consultation de II avec l'indice -1 (lorsque $x_1 = 0$ ou $y_1 = 0$), utiliser la valeur 0.

9.14.3.3 🧠 Fondement théorique

Élément	Rôle dans le détecteur Haar Cascade
Image intégrale II	Précalculée une seule fois par image, en temps $O(HW)$
Requête en O(1)	Chaque caractéristique Haar (différence entre les sommes de régions rectangulaires) est évaluée avec peu d'opérations, quelle que soit l'aire du rectangle
Scalabilité	C'est cette constance qui rend possible l'évaluation de milliers de caractéristiques, à de multiples positions et échelles, en temps réel
Principe d'inclusion-exclusion	Les quatre termes de la formule additionnent la région souhaitée et soustraient exactement les zones comptées en double

9.14.3.4 📦 Spécification des entrées et sorties (VPL)

Entrée :

- Ligne 1 : Entiers H et W.
- H lignes suivantes : W entiers chacune (image originale).
- Ligne suivante : Entier Q.
- Q lignes suivantes : quatre entiers x_1 y_1 x_2 y_2.

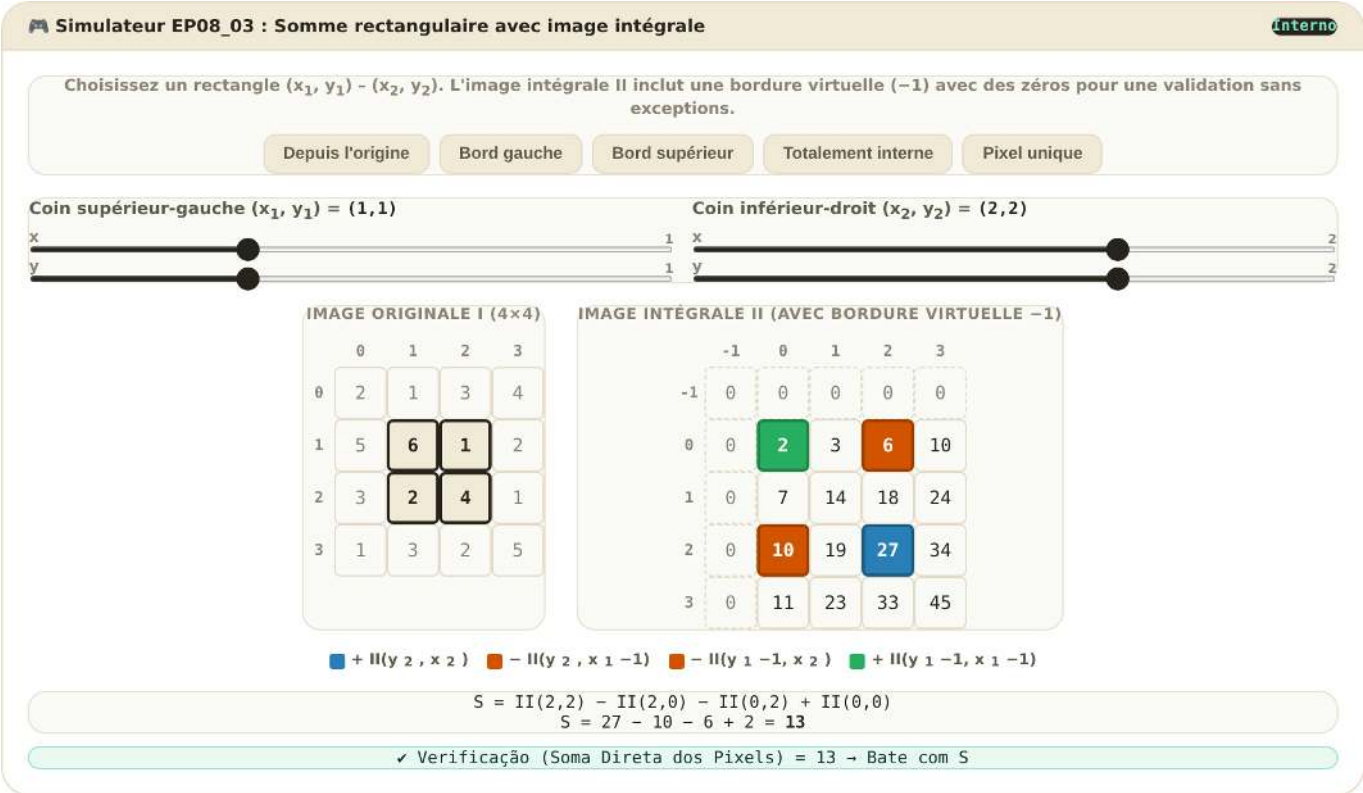
Sortie :

- H lignes avec W entiers chacune (image intégrale).
- Q lignes, une par requête, avec la somme de la région correspondante.

9.14.3.5 📌 Exemples

Entrée	Sortie	Observation
3 3 1 2 3 4 5 6 7 8 9 1 0 0 2 2	1 3 6 5 12 21 12 27 45 45	La requête couvre l'image entière ; la somme coïncide avec $II(2, 2)$ et avec la somme des 9 valeurs.

Entrée	Sortie	Observation
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 première requête utilise les quatre termes de la formule ; la seconde coïncide directement avec $II(1,1)$, car elle commence à l'origine.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap08/sim-ep0803.html

Figure 9.17: Simulateur EP08_03 : Somme Rectangulaire en O(1) — Multiples Situations de Bord

```
%%writefile EP08_03.py
# Code Python
```

```
Overwriting EP08_03.py
```

```
TestSuite("EP08_03.py").run()
```

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

9.14.4 EP08_04 🟢 IoU et Suppression des Non-Maximums (NMS)

La figure de cette section a montré l'effet de la Suppression des Non-Maximums sur un ensemble de boîtes produites par un détecteur de type *sliding window* : de multiples détections redondantes par objet ont été réduites à une seule boîte par objet. Vous avez été chargé de réimplémenter, octet par octet, les deux fonctions qui ont produit ce résultat — `calcular_iou` et `supressao_nao_maximos` — afin de confirmer, de vos propres mains, exactement les nombres que le chapitre a présentés.

9.14.4.1 📋 Directives d'implémentation

1. **Entrée :** Lire l'entier N (nombre de boîtes) et le réel τ (seuil d'IoU). Ensuite, lire N lignes, chacune avec cinq réels x_{min} y_{min} x_{max} y_{max} score.
2. **Intersection sur Union :** Pour deux boîtes A et B,

$$\text{IoU}(A, B) = \frac{\text{aire}(A \cap B)}{\text{aire}(A \cup B)},$$

   avec une aire d'intersection nulle lorsque les boîtes ne se chevauchent pas.
3. **Algorithme de NMS** (exactement comme décrit dans le chapitre) :
   a. Trier les boîtes par `score` décroissant (les égalités conservent l'ordre de lecture original).
   b. Sélectionner la boîte ayant le score le plus élevé parmi les restantes ; l'ajouter à la sortie et la retirer de la liste.
   c. Éliminer, de la liste restante, **toutes** les boîtes dont l'IoU avec la boîte sélectionnée est **supérieur ou égal** à τ — seules les boîtes avec IoU $< \tau$ restent candidates.
   d. Répéter (b)–(c) jusqu'à ce que la liste des restantes soit vide.
4. **Sortie :** Pour chaque boîte conservée, dans l'ordre où elle a été sélectionnée, imprimer son indice original (position de lecture, à partir de 0) et son `score`, avec 2 décimales. À la fin, imprimer `Total mantidas: X`.

9.14.4.2 📌 Contraintes computationnelles

- **Attention au sens du seuil :** contrairement à ce que l'on pourrait supposer, une boîte est **supprimée** lorsque IoU $\geq \tau$ (et non seulement lorsque IoU $> \tau$) — suivez exactement ce critère, le même que celui du code de référence du chapitre.
- **Indices originaux :** la sortie référence la position de lecture de chaque boîte dans l'entrée, et non sa position après le tri par `score`.
- **Aire sans ajout de 1 pixel :** utilisez l'aire $= (x_{max} - x_{min}) \times (y_{max} - y_{min})$, exactement comme dans le chapitre (sans l'ajustement « +1 » parfois utilisé dans d'autres conventions).

9.14.4.3 🧠 Fondements théoriques

Élément	Rôle dans le post-traitement
IoU	Quantifie le chevauchement spatial entre deux boîtes englobantes
Sliding window (Haar Cascade)	Produit généralement plusieurs détections chevauchantes pour le même objet, à des positions et échelles proches
Seuil τ	Contrôle l'agressivité de la suppression : trop faible fusionne les objets proches ; trop élevé laisse passer les redondances
Tri par `score`	Garantit que, parmi les boîtes redondantes, celle avec la plus grande confiance survive toujours

9.14.4.4 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

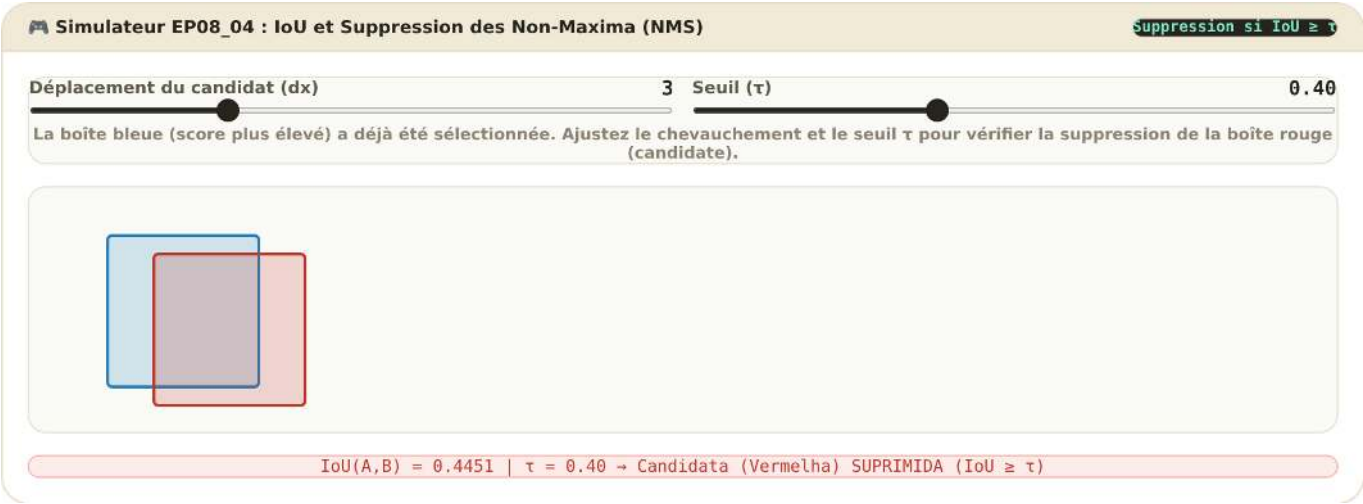
- Ligne 1 : Entier N et réel τ.
- Lignes suivantes N : cinq réels x_{min} y_{min} x_{max} y_{max} score.

Sortie :

- Une ligne par boîte conservée, dans l'ordre de sélection : `indice score` (score avec 2 décimales).
- Dernière ligne : `Total mantidas: X`.

9.14.4.5 📌 Exemples

Entrée	Sortie	Observation
5 0.4 50 50 150 150 0.90 60 55 155 145 0.75 58 60 160 150 0.60 300 300 400 420 0.95 310 305 395 415 0.70	3 0.95 0 0.90 Total mantidas: 2	Exactement l'exemple de la figure du chapitre : 5 boîtes redondantes (2 objets) deviennent 2 détections finales. L'IoU entre la 1re et la 2e boîtes est $\approx 0{,}775$, bien au-dessus de $\tau = 0{,}4$.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap08/sim-ep0804.html

Figure 9.18: Simulateur EP08_04: IoU et suppression des non-maximums

```
%%writefile EP08_04.py
# Code Python
```

```
Overwriting EP08_04.py
```

```
TestSuite("EP08_04.py").run()
```

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

9.14.5 EP08_05 🟡 Étiquetage des Composantes Connexes : Segmentation d'Instances

L'exemple de segmentation classique de ce chapitre a séparé les « instances » de pièces simplement par leur déconnexion spatiale dans le masque binaire résultant du seuillage d'Otsu. Cette étape finale — étiqueter chaque composante connexe avec un identifiant d'instance — est exactement ce que vous êtes chargé d'implémenter ici, à partir de zéro, sur un masque binaire déjà prêt (0 = fond, 1 = objet), comme s'il s'agissait d'une réimplémentation manuelle de `cv2.connectedComponents`.

Cet exercice met également en évidence, de manière très concrète, la limitation discutée dans le chapitre : le résultat dépend entièrement de la façon dont on définit la « voisinage » entre pixels — et, comme vous le verrez dans le deuxième exemple, deux pixels en diagonale peuvent être considérés comme la même instance

ou comme des instances différentes, en fonction exclusivement de la **connectivité** choisie, et non d'une quelconque notion sémantique d'objet.

9.14.5.1 📋 Directives d'Implémentation

1. **Entrée :** Lire les dimensions $H \times W$ du masque binaire et ses $H \times W$ valeurs (0 ou 1).
2. **Connectivité :** Lire l'entier $c \in \{4, 8\}$. En connectivité 4, les voisins de (i, j) sont $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ et $(i, j+1)$. En connectivité 8, on ajoute les quatre diagonales : $(i-1, j-1)$, $(i-1, j+1)$, $(i+1, j-1)$ et $(i+1, j+1)$.
3. **Découverte des composantes :** En parcourant le masque ligne par ligne, de gauche à droite et de haut en bas, chaque fois qu'un pixel de valeur 1 encore non étiqueté est trouvé, il initie une **nouvelle composante** : attribuez-lui le prochain étiquette disponible (la première composante découverte reçoit l'étiquette 1, la deuxième l'étiquette 2, et ainsi de suite) et propagez cette même étiquette à tous les pixels de valeur 1 atteignables à partir de lui par une chaîne de voisins (selon la connectivité choisie) — par recherche en largeur, en profondeur, ou par *union-find*, à votre choix.
4. **Pixels de fond :** restent avec l'étiquette 0 et n'appartiennent à aucune instance.
5. **Sortie :** D'abord, imprimer la carte complète des étiquettes — H lignes avec W entiers chacune. Ensuite, pour chaque étiquette ℓ de 1 à K (dans l'ordre de découverte), imprimer `Instance l: A pixels`, où A est la quantité de pixels avec cette étiquette. Enfin, imprimer `Total d'instances: K`.

9.14.5.2 📌 Contraintes Computationnelles

- **Ordre de découverte = ordre de parcours :** les étiquettes sont numérotées dans l'ordre où chaque nouvelle composante est trouvée par le parcours ligne par ligne, et non par taille ou position.
- **Connectivité explicite :** deux pixels de valeur 1 n'appartiennent à la même instance que s'il existe une chaîne de voisins **selon** c les reliant l'un à l'autre — ne pas utiliser par erreur la connectivité opposée.
- **Masque binaire pur :** toutes les valeurs d'entrée sont exactement 0 ou 1.

9.14.5.3 🧠 Fondement Théorique

Élément	Rôle dans la segmentation classique d'instances
Seuillage (Otsu, Chap. 4)	Étape précédente qui produit le masque binaire à partir de l'image d'intensité
Composante connexe	Chaque instance est définie **uniquement** par la connectivité spatiale des pixels d'objet, sans aucune notion de forme, de classe ou d'apparence
Connectivité 4 vs. 8	Paramètre qui modifie le résultat : en connectivité 8, deux blobs unis uniquement en diagonale deviennent une seule instance
Limitation centrale	La technique fusionne des instances qui se touchent ou se chevauchent (même si ce sont des objets clairement distincts), car il n'y a pas de notion d'« objet » — seulement de « région connectée »

9.14.5.4 📦 Spécification d'Entrée et de Sortie (VPL)

Entrée :

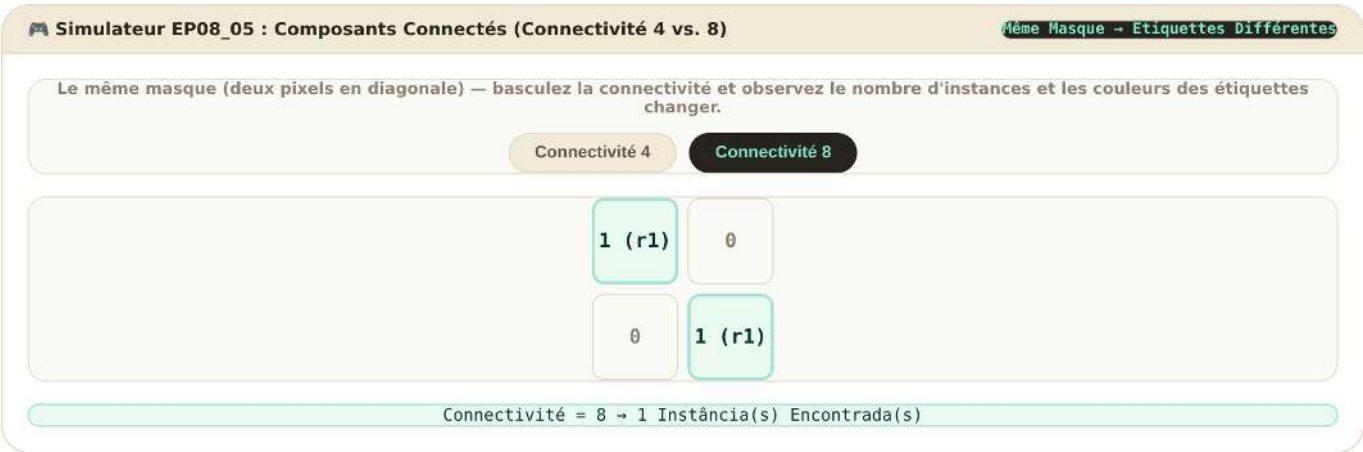
- Ligne 1 : Entiers H et W.
- Les H lignes suivantes : W entiers (0 ou 1) chacune.
- Dernière ligne : Entier c (4 ou 8).

Sortie :

- H lignes avec W entiers chacune (la carte des étiquettes).
- Une ligne par instance, dans l'ordre de découverte : `Instance l: A pixels`.
- Dernière ligne : `Total d'instances: K`.

9.14.5.5 📌 Exemples

Entrée	Sortie	Observation
6 6 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 8	0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 0 0 0 0 2 2 Instance 1: 4 pixels Instance 2: 4 pixels Total d'instances: 2	Deux blocs 2×2 clairement séparés : le résultat est le même en connectivité 4 ou 8.
2 2 1 0 0 1 8	1 0 0 1 Instance 1: 2 pixels Total d'instances: 1	En connectivité 8, les deux pixels en diagonale appartiennent à la **même** instance. Répétez cet exemple avec $c = 4$: le résultat devient 2 instances de 1 pixel chacune — uniquement par le changement de connectivité, sans aucune différence dans le masque.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap08/sim-ep0805.html

Figure 9.19: Simulateur EP08_05 : Étiquetage des composants connexes — Connectivité 4 vs. 8

```
%%writefile EP08_05.py
# Code Python
```

```
Overwriting EP08_05.py
```

```
TestSuite("EP08_05.py").run()
```

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

9.14.6 EP08_06 🟡 *Bounding Boxes*, Centroïdes et Propriétés des Instances avec `mm.measure`

Dans l'exercice précédent (**EP08_05**), on peut observer comment la segmentation par composantes connexes étiquette des régions binaires contiguës pour séparer les instances. Cependant, pour les tâches de détection, de suivi et d'analyse quantitative d'objets, la simple carte d'étiquettes ne suffit pas. Il devient nécessaire d'extraire des **métriques spatiales et géométriques** qui caractérisent chaque instance individuellement.

Cet EP se concentre sur le calcul et l'extraction automatique des propriétés fondamentales de vision par ordinateur pour chaque composante connexe trouvée dans le masque binaire, en utilisant la méthode native `mm.measure(img)` de la bibliothèque `morph` :

1. **Boîte englobante (*Bounding Box*) :** Le plus petit rectangle aligné sur les axes qui englobe complètement l'instance, défini par son coin supérieur gauche (x, y), sa largeur w et sa hauteur h.
2. **Centroïde géométrique** $(\bar{x}, \bar{y})$ **:** Le centre de masse de l'instance sur la grille discrète, équivalent aux moments spatiaux du premier ordre M_{10}/M_{00} et M_{01}/M_{00}.
3. **Aire géométrique du contour** (A) **:** L'aire délimitée par le contour de l'instance calculée via `mm.contourArea(c)`.

9.14.6.1 📋 Directives d'implémentation

1. **Entrée :** Lire les dimensions $H \times W$ du masque binaire, les $H \times W$ valeurs (0 ou 1) et le paramètre de connectivité $c \in \{4, 8\}$.
2. **Extraction automatique avec `mm.measure` :** Passer l'image binarisée à la fonction `mm.measure(img_bin)`, qui extrait les contours OpenCV et retourne une liste de dictionnaires contenant les propriétés géométriques de chaque instance.
3. **Propriétés retournées :** Pour chaque dictionnaire m de la liste retournée par `medidas = mm.measure(img_bin)` :
   - **Aire (`area`) :** Valeur numérique de l'aire géométrique du contour `mm.contourArea(c)`.
   - **Boîte englobante (`bbox`) :** Tuple (x, y, w, h) représentant le coin supérieur gauche, la largeur et la hauteur.
   - **Centroïde (`center`) :** Tuple (c_x, c_y) avec les coordonnées du centre de masse M_{10}/M_{00} et M_{01}/M_{00}. Formater avec **deux décimales**.
4. **Sortie :** Pour chaque instance $1, \ldots, K$ trouvée (triée par ordre de découverte/position dans l'image), imprimer une ligne contenant ses propriétés. Enfin, imprimer le nombre total d'instances.
   - Pour trier, utiliser `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`.

9.14.6.2 🧠 Fondements théoriques

Propriété dans `mm.measure`	Calcul mathématique / Logique discrète	Application pratique en vision
`bbox` (OpenCV)	$[x, y, w, h] = [\min(c), \min(r), \Delta c + 1, \Delta r + 1]$	Format classique d'OpenCV. *Remarque : des réseaux comme YOLO convertissent ce rectangle en (c_x, c_y, w, h) normalisé.*
`center`	$\bar{x} = \frac{M_{10}}{M_{00}}, \quad \bar{y} = \frac{M_{01}}{M_{00}}$	Centre de masse exact du masque (utilisé pour le suivi et l'analyse de trajectoire).
`area`	$A = \text{contourArea}(C)$ (Formule du polygone)	Métrique continue de la surface de l'objet.

9.14.6.3 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entiers H et W.
- H lignes suivantes : W entiers (0 ou 1) chacun.
- Dernière ligne : Entier c (4 ou 8).

Sortie :

- Une ligne par instance dans l'ordre de découverte : `Instância l: Area=A, BBox=(x,y,w,h), Centroide=(cx,cy)`
- Dernière ligne : `Total de instâncias: K`.

9.14.6.4 📌 Exemples

Entrée	Sortie	Observation
6 6 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 8	Instância 1: Area=1.0, BBox=(1,1,2,2), Centroide=(1.50,1.50) Instância 2: Area=1.0, BBox=(4,4,2,2), Centroide=(4.50,4.50) Total de instâncias: 2	Blocs 2×2 alignés. Le calcul de l'aire géométrique du contour donne 1.0. Le centroïde du bloc dans les colonnes 1–2 et les lignes 1–2 est exactement (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	Instância 1: Area=2.0, BBox=(1,1,4,2), Centroide=(2.40,1.20) Total de instâncias: 1	Objet asymétrique en forme de "T" inversé. L'aire géométrique du contour est 2.0. Le centroïde reflète la distribution des pixels de l'objet.

```
%%writefile EP08_06.py
# Code Python
```

```
Overwriting EP08_06.py
```

```
TestSuite("EP08_06.py").run()
```

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

9.14.7 EP08_07 🟡 Suppression du Bruit Sel et Poivre et Mesure d'Objets

Dans cet exercice, vous appliquerez un filtrage morphologique pour nettoyer une image binaire corrompue par un bruit de type **sel et poivre** (pixels isolés de valeur `1` dans le fond et `0` à l'intérieur des objets). Après le nettoyage, le programme doit extraire les mesures géométriques des composants connectés restants, les trier et afficher le tableau final de métriques.

9.14.7.1 📋 Directives d'Implémentation

1. **Entrée :** lire deux entiers H et W (hauteur et largeur de l'image) sur la première ligne, puis les H lignes avec la matrice binaire contenant des pixels `0` et `1` séparés par des espaces.
2. **Filtrage Morphologique :** appliquer un enchaînement d'**Ouverture** (pour éliminer le bruit de sel dans le fond) suivi d'une **Fermeture** (pour combler le bruit de poivre à l'intérieur des objets) avec un élément structurant 3×3.
3. **Affichage de l'Image Nettoyée :** imprimer la matrice résultante en valeurs `0` et `1` séparées par des espaces.

Simulateur EP08_06 : Métriques Morphologiques Natives (mm.measure) Contour OpenCV & Moments

ACTION

Générer de nouvelles instances binaires

Paramètre de précision (approxPolyDP) : precision = 0.01

CARTE DES ÉTIQUETTES D'INSTANCES

0	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	0	0	0	0	0	0	0	0	0	0	0	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	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	0	0	0	0	0	0	0	0	2	2	2	2	2	0	0	0
0	0	0	1	1	1	0	3	3	3	0	0	0	0	2	0	2	2	2	0	0	0
0	0	0	0	0	0	0	3	3	3	0	0	0	0	0	2	2	2	2	0	0	0
0	0	0	0	0	0	0	0	3	3	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

MÉTRIQUES EXTRAITES PAR LE MM.MEASURE

id	aire	périmètre	centre (cx, cy)	bbox (x,y,w,h)	circularité	solidité	sommets
1	8.0	12.00	(4.00, 3.00)	(3, 1, 3, 5)	0.698	1.000	6
2	6.5	13.41	(16.23, 4.92)	(14, 4, 5, 3)	0.454	0.867	6
3	3.5	7.41	(8.13, 5.88)	(7, 5, 3, 3)	0.800	1.000	5

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap08/sim-ep0806.html

Figure 9.20: Simulador EP08_06 : Extraction de *Bounding Boxes*, Centroides et Propriétés avec *mm.measure*

4. **Mesures Géométriques :** pour chaque objet identifié dans la matrice nettoyée, extraire :

- `id` : identifiant numérique séquentiel (réattribué après le tri) ;
- `area` : aire calculée via le contour (`cv2.contourArea`) ;
- `perimeter` : périmètre du contour (`cv2.arcLength`) ;
- `cx`, `cy` : centre de masse (centroïde via `cv2.moments`) ;
- `x`, `y`, `w`, `h` : coordonnées du rectangle englobant (`cv2.boundingRect`) ;
- `circularity` : circularité donnée par $\frac{4\pi \cdot \text{aire}}{\text{périmètre}^2}$;
- `solidity` : solidité donnée par le rapport $\frac{\text{aire}}{\text{aire de l'enveloppe convexe}}$;
- `vertices` : nombre de sommets approximé du polygone (`cv2.approxPolyDP` avec $\epsilon = 0.02 \times$ périmètre).

5. **Tri et Sortie :** trier les objets par ordre croissant selon la position X du rectangle englobant (`bbox[0]`) ; en cas d'égalité, utiliser la position Y (`bbox[1]`). Réattribuer les `ids` de 1 à N et imprimer le tableau formaté.
   - Pour trier, utiliser `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, avec `medidas = mm.measure(img)`.

9.14.7.2 📌 Contraintes et Règles de Tri

- **Règle de Tri des Objets :**

```
medidas.sort(key=lambda m: (m['bbox'][0], m['bbox'][1]))
```

- **Différence d'Aire :** L'aire calculée par OpenCV (`cv2.contourArea`) mesure l'aire du polygone continu délimité par les centres des pixels de bordure, ce qui donne des valeurs numériques inférieures au simple comptage discret des pixels `1` (`np.sum`).

9.14.7.3 🧠 Fondements Théoriques

Opération / Métrique	Fonction dans le Filtrage et la Caractérisation
Ouverture Morphologique ($\circ$)	Érosion suivie d'une dilatation : supprime les bruits brillants isolés (*sel*).
Fermeture Morphologique ($\bullet$)	Dilatation suivie d'une érosion : comble les petits trous sombres à l'intérieur des objets (*poivre*).
`cv2.boundingRect`	Retourne (x, y, w, h), le plus petit rectangle aligné sur les axes qui englobe l'objet.
Circularité et Solidité	Décrivent la compacité et la convexité géométrique du composant.

9.14.7.4 📌 Exemples

Entrée	Sortie
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
# Code Python
```

```
Overwriting EP08_07.py
```

```
TestSuite("EP08_07.py").run()
```

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

9.14.8 EP08_08 🟡 Image en Niveaux de Gris et Seuillage Dynamique

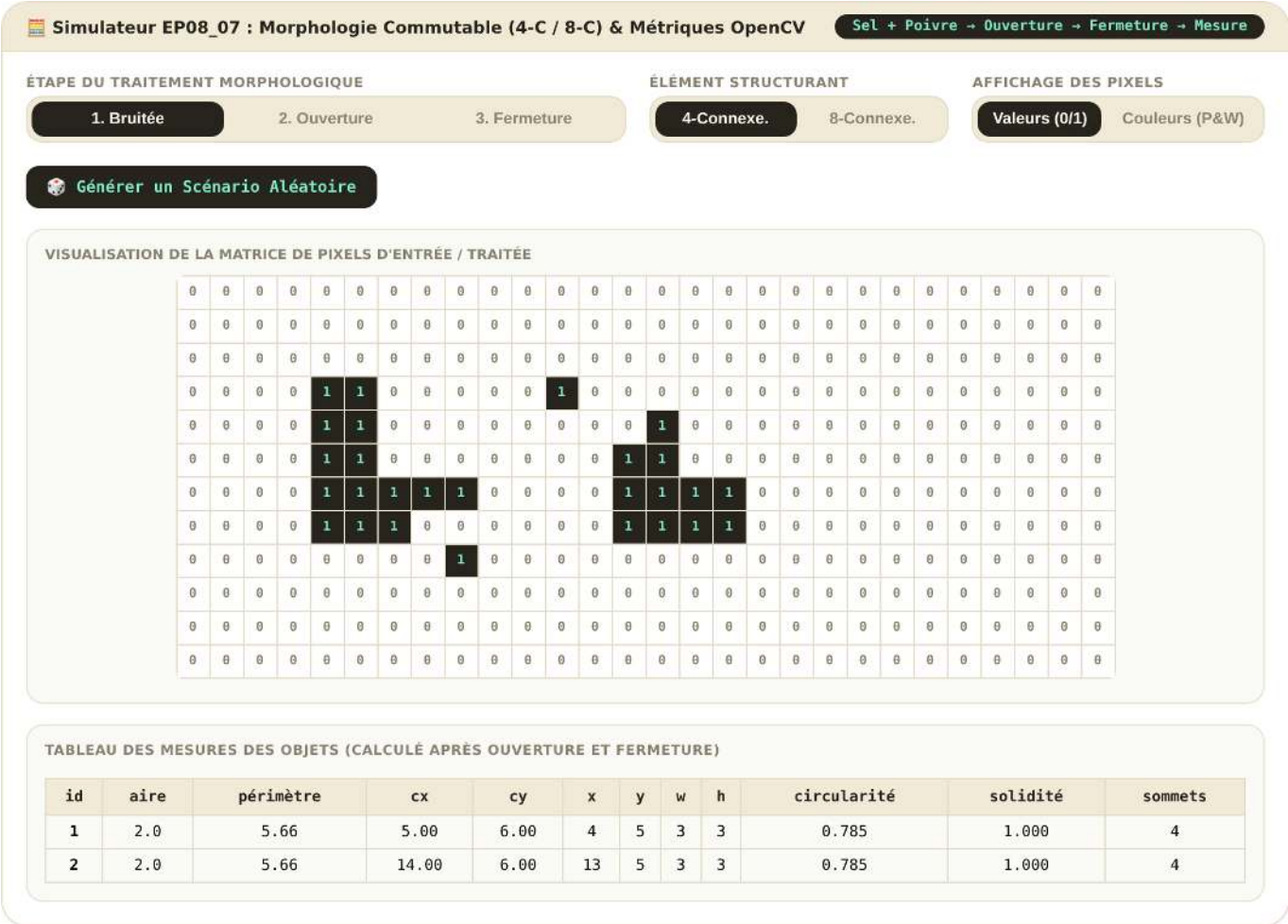
Dans cet exercice, l'image d'entrée n'est plus strictement binaire (`0/1`) mais devient une **image en niveaux de gris (8 bits,** $0 \ldots 255$**)**, où les objets possèdent une intensité moyenne intermédiaire sur un fond sombre (0), avec en plus un bruit de type sel et poivre réparti sur toute l'image.

9.14.8.1 📋 Directives d'Implémentation

1. **Entrée :** lire H et W sur la première ligne, suivis des H lignes contenant des valeurs entières de 0 à 255 dans une matrice $H \times W$.
2. **Prétraitement :**

- Appliquer un filtre **Médian** (3×3) pour éliminer le bruit sel et poivre tout en préservant des bords nets.
- Appliquer un **Seuillage d'Otsu** (ou un seuil fixe $T = 60$) pour binariser l'image nettoyée.

3. **Mesure et Sortie :** extraire le contour des objets, calculer les métriques géométriques (`area`, `perimeter`, `cx`, `cy`, `x`, `y`, `w`, `h`, `circularity`, `solidity`) et trier les objets par `bbox[0]` (et `bbox[1]` en cas d'égalité). Réattribuer l'`id` de 1 à N et imprimer le tableau.



id	aire	périmètre	cx	cy	x	y	w	h	circularité	solidité	sommets
1	2.0	5.66	5.00	6.00	4	5	3	3	0.785	1.000	4
2	2.0	5.66	14.00	6.00	13	5	3	3	0.785	1.000	4

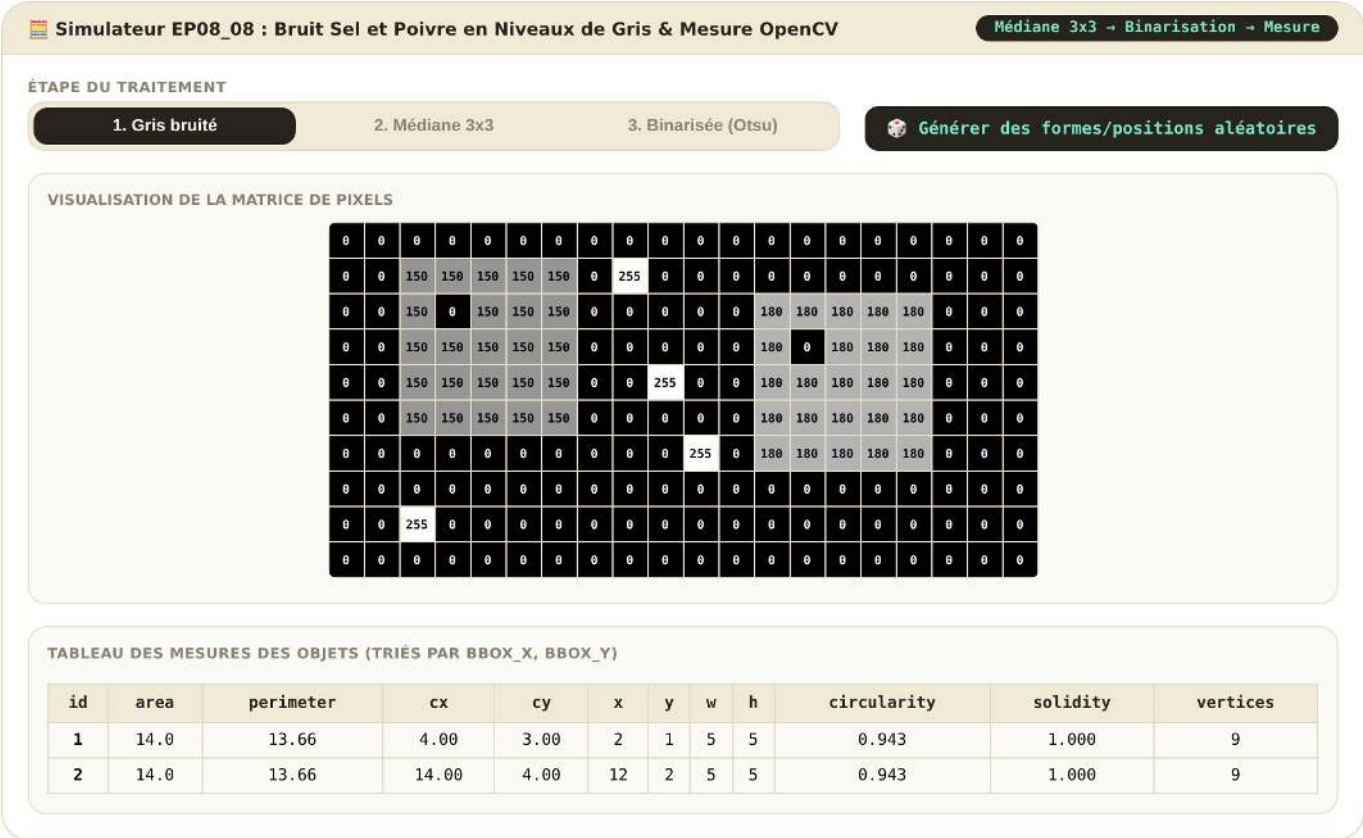
Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap08/sim-ep0807.html

Figure 9.21: Simulateur EP08_07: Morphologie avec Connectivité Configurable et Mesure

- Pour le tri, utiliser `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, avec `medidas = mm.measure(img)`.

9.14.8.2 📌 Exemples

Entrée	Sortie
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 … (image binaire contenant un carré et un cercle)	id area perimeter cx cy x y w h circularity solidity vertices 1 16.0 16.0 8.0 8.0 6 6 5 5 0.79 1.000 4 2 28.3 18.8 22.5 8.0 19 5 7 7 1.00 1.000 8



id	area	perimeter	cx	cy	x	y	w	h	circularity	solidity	vertices
1	14.0	13.66	4.00	3.00	2	1	5	5	0.943	1.000	9
2	14.0	13.66	14.00	4.00	12	2	5	5	0.943	1.000	9

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap08/sim-ep0808.html

Figure 9.22: Simulateur EP08_08 : Filtrage Médian en Niveaux de Gris et Mesure d'Objets OpenCV

```
%%writefile EP08_08.py
# Code Python
```

```
Overwriting EP08_08.py
```

```
TestSuite("EP08_08.py").run()
```

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

9.14.9 EP08_09 🟠 Gradient d'éclairage et seuillage adaptatif

Dans cette variante, les objets sont immergés dans un fond avec un **éclairage non uniforme (gradient doux d'éclairage)**. Le seuillage simple par valeur unique échoue, nécessitant un prétraitement plus robuste.

9.14.9.1 📋 Directives d'implémentation

1. **Entrée :** image en niveaux de gris $H \times W$ avec une variation de fond de 20 à 180.
2. **Prétraitement :**

- Appliquer un **seuillage adaptatif** (ex. : `cv2.adaptiveThreshold` avec une fenêtre gaussienne de 15×15 et une constante $C = 3$) pour isoler les objets indépendamment de la variation du fond.

  `cv2.adaptiveThreshold(img_gray, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, ksize, C) // 255`

  `ksize` et `C` sont lus après l'image.

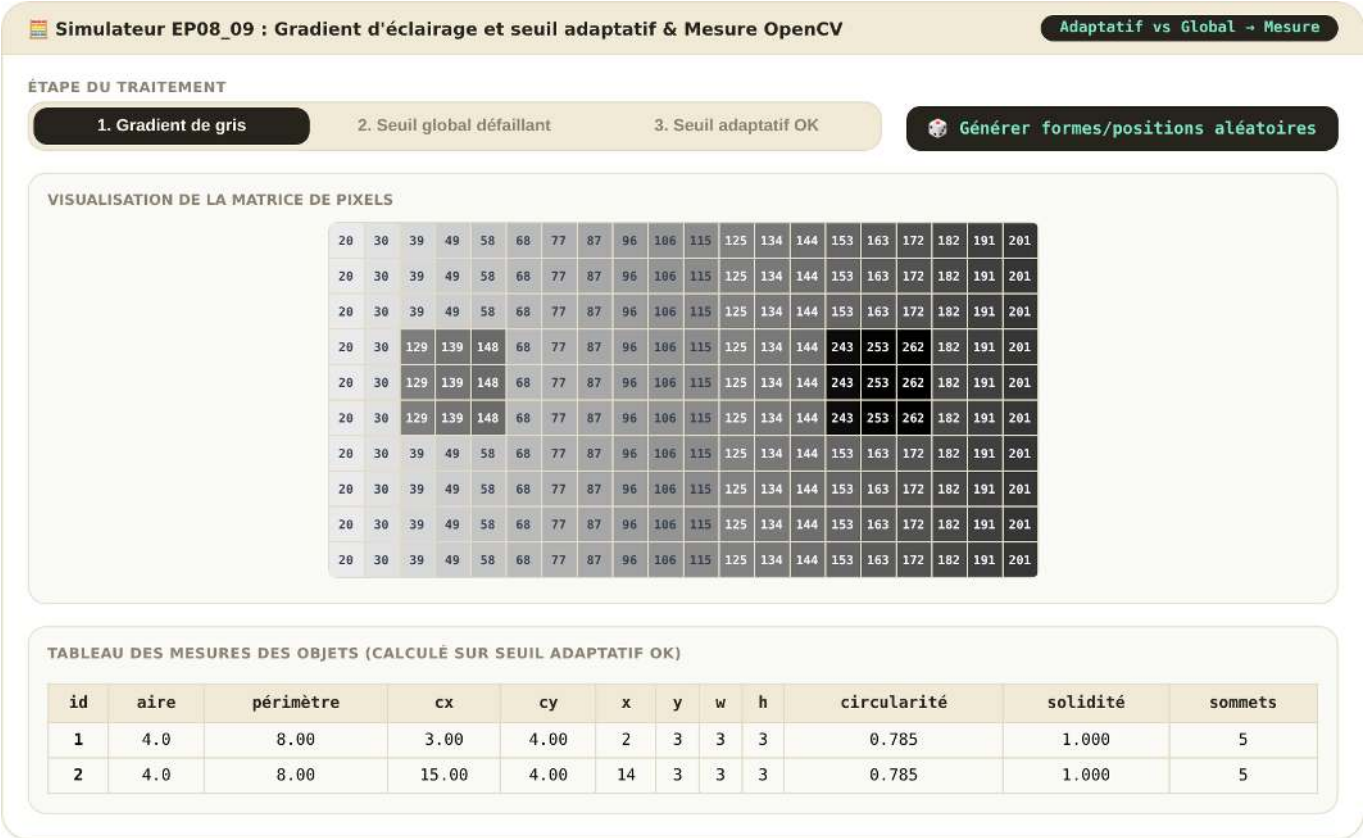
- Opération morphologique de **fermeture** (3×3) pour sceller d'éventuelles défaillances dans les contours.

1. **Mesure et classification :** extraire les mesures.

2. **Tri et sortie :** trier par `(bbox[0], bbox[1])` et imprimer le tableau incluant la colonne `class`.
   - Pour trier, utiliser `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, avec `medidas = mm.measure(img, precision=0.02)`.

9.14.9.2 📌 Exemples

Entrée	Sortie
16 32 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 … k 20	id area perimeter cx cy x y w h circularity solidity vertices 1 9.0 12.0 10.0 5.0 8 3 5 5 0.79 1.000 4 2 28.3 18.8 25.0 12.0 22 9 7 7 1.00 1.000 3



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap08/sim-ep0809.html

Figure 9.23: Simulateur EP08_09 : Gradient d'éclairage, seuil adaptatif et mesure OpenCV

```
%%writefile EP08_09.py
# Code Python
```

```
Overwriting EP08_09.py
```

```
TestSuite("EP08_09.py").run()
```

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

9.14.10 EP08_10 🔴 Contraste Faible et Séparation d'Objets Thèmes (*Watershed* / Distance)

Dans cet exercice, **certains objets géométriques sont légèrement en contact (superposés sur les bords)**. La simple extraction des contours traiterait deux objets comme s'ils n'en formaient qu'un seul.

9.14.10.1 📋 Directives d'Implémentation

1. **Entrée :** matrice $H \times W$ en niveaux de gris avec des objets d'intensité $110 \ldots 140$ sur un fond 0, avec bruit et une paire d'objets tangents.
2. **Prétraitement et Séparation :**

- Application du seuillage.
- Application de la **Transformée de Distance** (`mm.dist`).
- Obtention des pics de distance pour servir de marqueurs dans la **Transformée *Watershed*** (`mm.watershed`), séparant physiquement les objets en contact dans le masque. **Astuce :** utiliser `mm.regmax()` pour obtenir les maxima locaux, puis les étiqueter avec `mm.label0`.
- Après le *watershed*, appliquer à nouveau le seuillage avec `mm.threshold(water,0)//255`.

3. **Analyse des Composants Connectés :** mesurer chaque région isolée après le *Watershed*.
4. **Sortie :** imprimer les composants triés par `(bbox[0], bbox[1])` avec leurs métriques individuelles de surface, centroïde et solidité.
   - Pour trier, utiliser `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, avec `medidas = mm.measure(img, precision=0.02)`.

9.14.10.2 📌 Exemples

Entrée	Sortie
16 16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 … (image binaire contenant deux carrés)	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

Simulateur EP08_10 : Séparation de disques tangents (Transformée L2 & Watershed)

mm.dist L2 → mm.watershed → Mesure

ÉTAPE DU TRAITEMENT MORPHOLOGIQUE

1. Gris bruité | 2. Masque uni | 3. Distance L2 | 4. Watershed (Coupe)

Générer des disques avec rayons aléatoires

VISUALISATION DE LA MATRICE DE PIXELS

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	140	0	0	0	0	0	147	0	0	0	0	0	0	0	0	0	0	0
0	141	158	153	141	152	0	147	144	149	140	157	0	0	0	0	0	0	0	0	0
0	151	152	158	150	158	0	157	141	151	155	150	0	0	0	0	0	0	0	0	0
143	147	157	140	148	140	154	153	154	141	158	159	142	0	0	0	0	0	0	0	0
0	141	150	143	151	157	0	142	147	143	152	156	0	0	0	0	0	0	0	0	0
0	149	157	152	151	143	0	146	153	151	153	144	0	0	0	0	0	0	0	0	0
0	0	0	145	0	0	0	0	0	141	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

TABLEAU DES MESURES DES DISQUES APRÈS LA COUPE DU WATERSHED

id	aire	périmètre	cx	cy	x	y	w	h	circularité	solidité	sommets
1	19.0	18.49	2.89	5.00	0	2	6	7	0.699	0.864	13
2	19.0	18.49	9.11	5.00	7	2	6	7	0.699	0.864	15

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap08/sim-ep0810.html

Figure 9.24: Simulateur EP08_10: Séparation de Disques Tangents via Transformée de Distance L2 et *Watershed*

```
%%writefile EP08_10.py
# Code Python
```

```
Overwriting EP08_10.py
```

```
TestSuite("EP08_10.py").run()
```

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

9.14.11 EP08_11 🔴 Classification et Validation d'Objets avec Gabarit de *Bounding Box*

Dans cet exercice, l'objectif est de traiter une image en niveaux de gris contenant plusieurs objets géométriques, d'extraire leurs propriétés avec `mm.measure` et de valider les boîtes englobantes (*bounding boxes*) détectées par rapport à un gabarit réel (*Ground Truth* - GT) fourni en entrée, en utilisant la métrique IoU (*Intersection over Union*).

9.14.11.1 📋 Directives d'implémentation

1. **Lecture de l'image :** Lire les dimensions $H \times W$ et la matrice $H \times W$ de pixels de l'image en niveaux de gris.
2. ***Pipeline* morphologique :** Binariser l'image via la méthode d'Otsu (`mm.threshold`) et afficher le masque binarisé résultant à l'aide de `mm.drawImg`.
3. **Lecture du gabarit réel (*Ground Truth*) :**

- Lire la quantité G de boîtes englobantes du gabarit.
- Si $G > 0$, lire G lignes contenant chacune 5 valeurs : `id xmin_norm ymin_norm xmax_norm ymax_norm`.
- **Conversion des coordonnées :** Les coordonnées du gabarit sont normalisées dans la plage $[0.0, 1.0]$. Pour les convertir en pixels dans la grille de l'image :

$$x_{\min} = \lfloor \text{xmin_norm} \times W \rfloor, \quad y_{\min} = \lfloor \text{ymin_norm} \times H \rfloor$$

$$w = \lfloor \text{xmax_norm} \times W \rfloor - x_{\min}, \quad h = \lfloor \text{ymax_norm} \times H \rfloor - y_{\min}$$

4. **Extraction des métriques et calcul de l'IoU :**

- Extraire les propriétés des instances avec `mm.measure(img_bin, precision=0.02)`.
- Pour chaque *bounding box* détectée (x, y, w, h), calculer le chevauchement IoU par rapport aux boîtes du gabarit et définir `hits = 1` s'il existe une correspondance (*match*) avec IoU ≥ 0.50, ou `hits = 0` dans le cas contraire.

5. **Sortie :** Trier les instances par position `(bbox[0], bbox[1])` et imprimer le tableau CSV avec la colonne supplémentaire `hits`.
   - Pour le tri, utiliser `medidas.sort(key=lambda m: (m['bbox'][1], m['bbox'][0]))`, avec `medidas = mm.measure(img)`.

9.14.11.2 🧠 Fondements théoriques et conversion

Concept	Formule / Opération	Description
BBox détectée	(x, y, w, h) via `mm.measure`	Boîte englobante calculée sur la grille discrète en pixels entiers.
BBox gabarit (GT)	$(x_{\min}, y_{\min}, w, h)$ convertis	Boîte réelle fournie en entrée en coordonnées relatives $[0.0, 1.0]$.

Concept	Formule / Opération	Description
IoU (Intersection over Union)	$\text{IoU} = \frac{\text{Aire}(B_{\text{DET}} \cap B_{\text{GT}})}{\text{Aire}(B_{\text{DET}} \cup B_{\text{GT}})}$	Évalue le taux de chevauchement des boîtes. Est considérée comme valide si IoU ≥ 0.50.
Statut de validation (`hits`)	1 si $\max(\text{IoU}) \geq 0.50$, sinon 0	Indicateur binaire de succès du détecteur par rapport au gabarit.

9.14.11.3 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- **Ligne 1 :** Entiers H et W (dimensions de la matrice).
- **Les H lignes suivantes :** W entiers (0 à 255) représentant l'image en niveaux de gris.
- **Ligne $H+2$:** Entier G (quantité de boîtes du gabarit véritable).
- **Les G lignes suivantes :** 5 valeurs numériques par ligne : `id xmin_norm ymin_norm xmax_norm ymax_norm` (où les coordonnées sont des valeurs flottantes entre 0.0 et 1.0).

Sortie :

1. Matrice binarisée affichée via `mm.drawImg(img_bin)`.
2. En-tête CSV : `id,area,perimeter,cx,cy,x,y,w,h,circularity,solidity,vertices,hits`
3. Une ligne CSV par objet détecté contenant ses propriétés formatées et l'indicateur `hits` (1 ou 0).

9.14.11.4 📌 Exemples

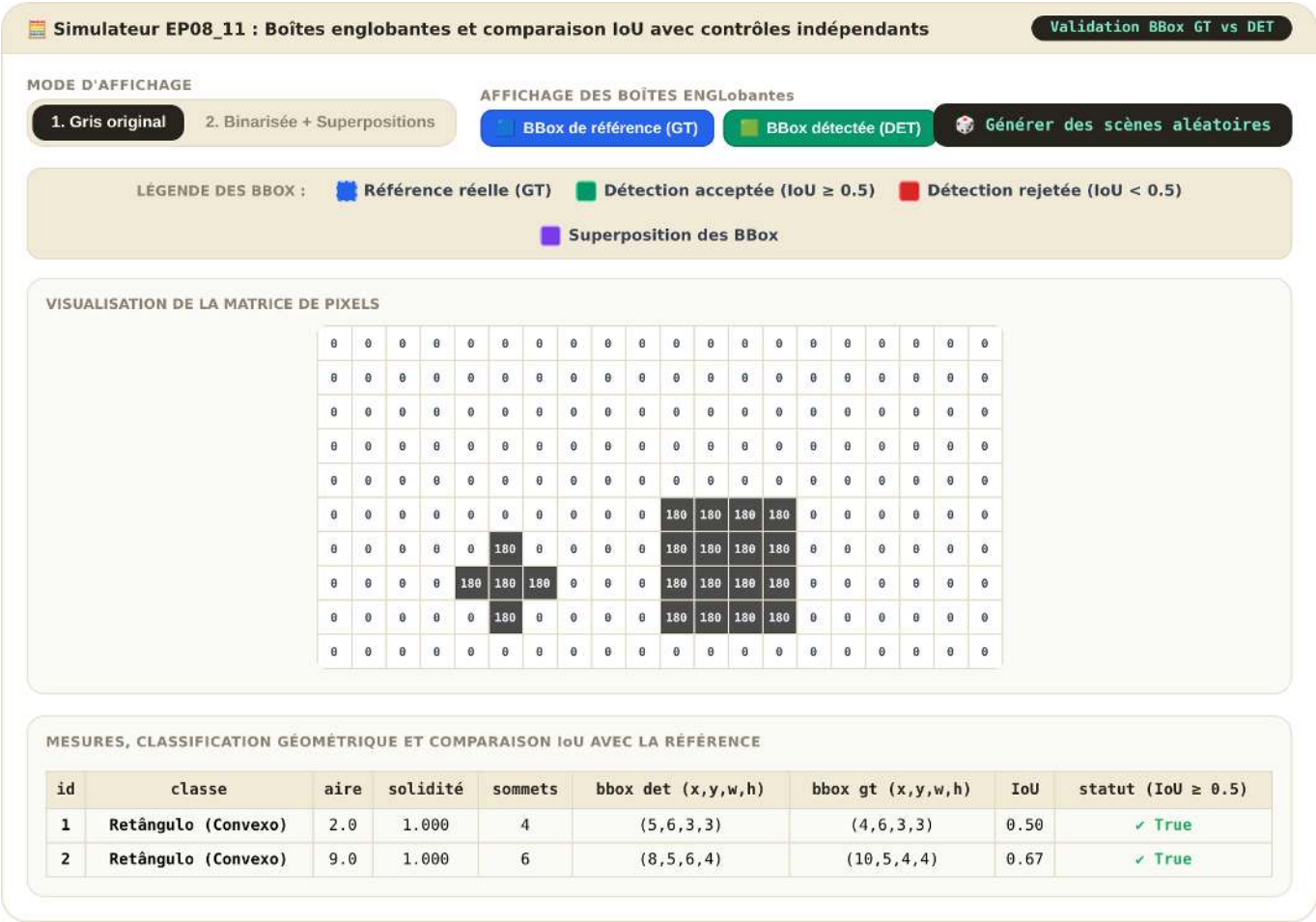
Entrée	Sortie
10 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 180 0 0 0 0 0 0 0 0 180 180 180 0 0 0 0 0 0 0 180 180 180 0 0 0 0 0 0 0 180 180 180 0 0 0 0 0 0 0 0 180 0 0 0 0 0 0 0 0 180 180 180 0 0 0 0 0 0 0 0 0 0 0 0 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
# Code Python
```

```
Overwriting EP08_11.py
```

```
TestSuite("EP08_11.py").run()
```

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



id	classe	aire	solidité	sommets	bbox det (x,y,w,h)	bbox gt (x,y,w,h)	IoU	statut (IoU ≥ 0.5)
1	Retângulo (Convexo)	2.0	1.000	4	(5,6,3,3)	(4,6,3,3)	0.50	✔ True
2	Retângulo (Convexo)	9.0	1.000	6	(8,5,6,4)	(10,5,4,4)	0.67	✔ True

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap08/sim-ep0811.html

Figure 9.25: Simulateur EP08_11 : Classification géométrique avec contrôles indépendants des *overlays BBox* (GT et DET)

9.14.12 EP08_12 🔴 Segmentation d'instances sur image réelle : objets géométriques

L'exemple de segmentation classique de ce chapitre a séparé des « instances » de pièces de monnaie par déconnexion spatiale dans le masque binaire résultant du seuillage d'Otsu. Dans cet exercice, vous appliquerez la même idée — mais cette fois sur une image réelle avec des objets géométriques variés — en enchaînant le prétraitement, la binarisation, l'extraction de contours (`cv2.findContours`) et la validation du résultat par rapport à une référence de *bounding boxes*.

Contrairement à l'exercice précédent (étiquetage sur un masque déjà prêt), ici vous partez de l'**image originale** : la qualité de votre segmentation dépend directement des choix de prétraitement (filtrage, seuillage, opérations morphologiques) effectués avant d'étiquetter les composants.

9.14.12.1 📋 Directives d'implémentation

1. **Entrée :** utiliser l'image `00000.jpg`.
2. **Prétraitement et segmentation :** appliquer les étapes nécessaires (filtrage, binarisation et opérations morphologiques) pour séparer automatiquement les objets du fond, sans recadrages manuels.
3. **Étiquetage et mesure :** pour chaque objet segmenté, déterminer :
   - l'aire ;
   - le centre de masse (centroïde) ;
   - le type, selon l'ensemble `obj2`.
4. **Annotation visuelle :** écrire, à l'intérieur de chaque objet, son aire et le sigle du type (`obj2`).
5. **Validation (IoU) :** calculer l'*Intersection over Union* (IoU) entre la *bounding box* détectée (`cv2.boundingRect`) et la *bounding box* de référence du type correspondant. Un objet est considéré comme correctement segmenté uniquement s'il existe exactement une *bounding box* du type correct avec **IoU 0,5**.
6. **Sortie :** imprimer, pour chaque objet détecté, son identifiant, son type et s'il a été validé avec succès (`acertou=1`) ou non. L'impression doit suivre l'ordre des classes de `obj2` (0=Tria … 8=Cruz) ; au sein de la même classe, trier les objets par coordonnée verticale du centroïde (`cy`) croissante. À la fin, imprimer la précision globale.

9.14.12.2 📌 Contraintes informatiques

- **Sans recadrage manuel :** toute la segmentation doit être effectuée sur l'image complète.
- **Ensemble de classes fixe :**

```
obj  = ['Triangulo','Quadrado','Pendagono','Hexagono','Heptagono','Circulo',
        'Elipse','Estrela','Cruz']
obj2 = ['Tria','Quad','Pent','Hexa','Hept','Circ','Elip','Estr','Cruz']
```

- **Dimensions de l'image :** 608×608 pixels — utilisées pour dénormaliser les coordonnées du fichier TXT.
- **Validation par centre de masse :** un objet n'est considéré comme correctement segmenté que si son centroïde se trouve strictement à l'intérieur de la *bounding box* de référence correspondant au même type d'objet.

9.14.12.3 🧠 Fondements théoriques

Élément	Rôle dans la segmentation d'instances
Prétraitement (filtrage, seuillage)	Étape qui produit le masque binaire à partir de l'image d'intensité originale
`cv2.findContours`	Extrait les contours des composants connectés dans le masque binaire

Élément	Rôle dans la segmentation d'instances
Moments géométriques (`cv2.moments`)	Permettent de calculer le centre de masse (centroïde) de chaque contour
`approxPolyDP` / sommets	Aide à la classification du type d'objet (nombre de côtés approximatif)
Validation via *bounding box*	Confirme si l'instance segmentée correspond spatialement à un objet de la référence, en mesurant la précision de la méthode

9.14.12.4 📌 Exemple de sortie attendue

```
Objeto 1: tipo=Tria, validado=True
...
Acurácia: 88.89%
```

Paramètres fixes pour la reproductibilité : pour que la sortie corresponde à la référence de correction automatique, utilisez exactement : filtre d'aire minimale de 300 pixels ; `cv2.approxPolyDP` avec `epsilon = 0.02 * périmètre` ; seuil de solidité 0,92 et nombre de sommets 9 (avec 11 pour différencier Cruz de Estrela) pour les formes concaves ; rapport d'aspect 1,15 pour différencier Círculo de Elipse ; seuil IoU = 0,5 lors de la validation.

9.14.12.5 📌 Fichiers de référence (`.jpg` et `.txt`)

Pour le débogage local, deux fichiers de référence sont fournis (inclus dans cette livraison ; lors de leur intégration au dépôt du chapitre, enregistrez-les dans `all/cap08/dados/EP08/`) :

- 📥 **Image (`00000.jpg`)** : image d'objets géométriques utilisée comme entrée de l'exercice. L'objectif est de segmenter automatiquement chaque objet, de déterminer son type et de calculer ses mesures.
- 📥 **Référence (`00000.txt`)** : fichier contenant les *bounding boxes* normalisées des objets présents dans l'image. Chaque ligne contient l'identifiant de la classe et les coordonnées normalisées des coins supérieur gauche et inférieur droit, utilisé pour valider automatiquement la segmentation.

La Figure 9.26 présente l'image d'entrée et la même image avec les *bounding boxes* dessinées à partir du fichier de référence.

```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"Téléchargement {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=[
        "Image originale",
        "Bounding boxes de la correction"
    ],
    cols=2,
    figsize=(10,5)
)
```

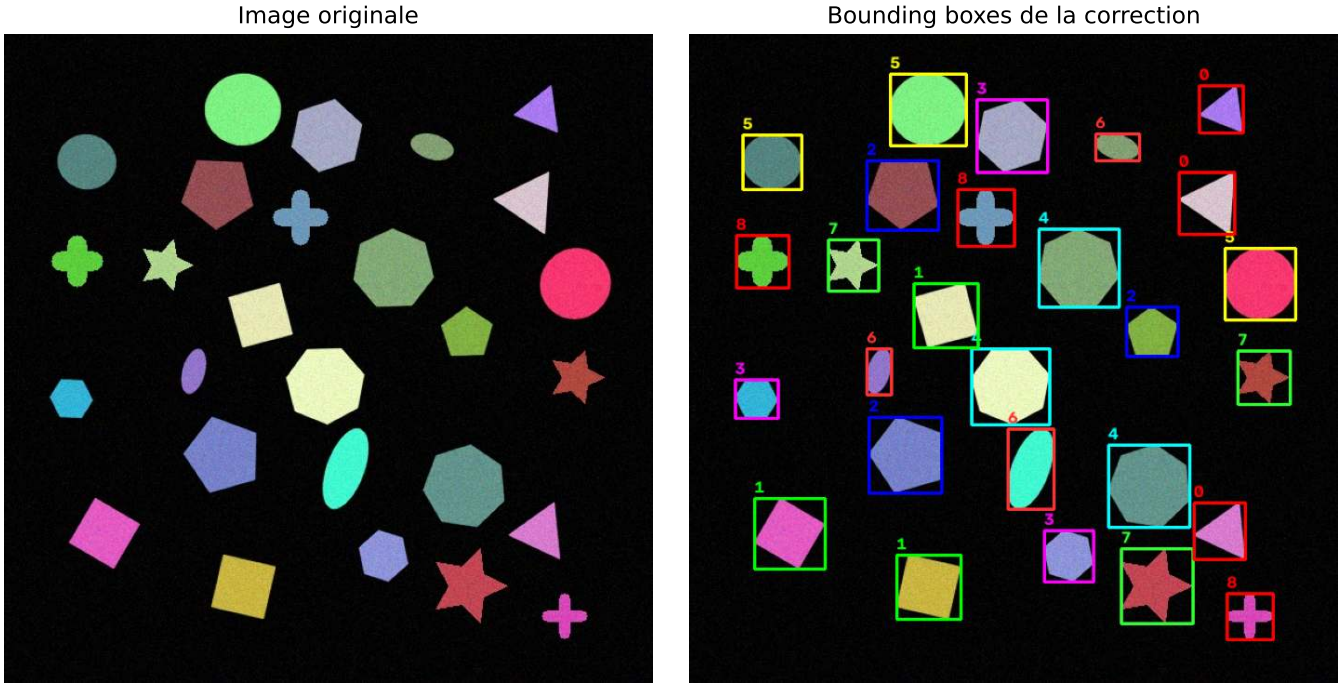


Figure 9.26: Simulateur EP08_12 : Image utilisée dans l'EP08_12. À gauche, l'image originale. À droite, l'image avec les *bounding boxes* du fichier de correction.

```
%%writefile EP08_12.py
# Code Python
```

```
Overwriting EP08_12.py
```

```
TestSuite("EP08_12.py").run()
```

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

Simulateur EP08_12 : Précision de segmentation sur objets multiples

réussi si IoU ≥ seuil ET bon type

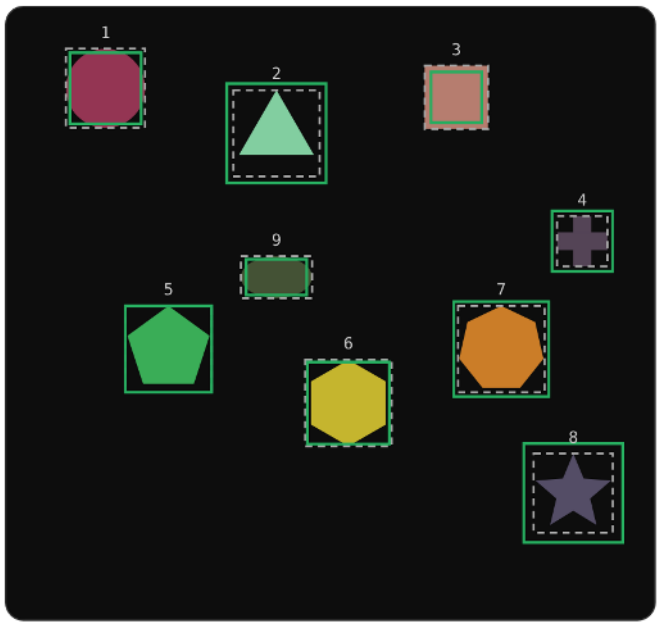
Chaque forme a une *boundbox* de référence (rectangle en pointillés, ajusté autour de la forme) et une *boundbox* détectée (rectangle plein, décalé/bruité). Ajustez le bruit, le biais et le seuil d'IoU pour voir la validation changer.

bruit de segmentation (px, jitter max par côté) 0

biais systématique en x (px) 0

seuil d'IoU 0.50

simuler une erreur de classification (2 objets avec type échangé)



id	type réel	type détecté	IoU	≥ seuil	réussi
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.fr/cap08/sim-ep0812.html

Figure 9.27: Simulateur EP08_12 : Précision de Segmentation sur Objets Multiples (IoU)

Chapitre 10

Apprentissage profond pour la vision par ordinateur

Exécuter Colab Ouvrir GitHub

Les chapitres précédents ont établi les fondements de la vision par ordinateur (VC) au moyen de méthodes classiques d'extraction et de représentation de caractéristiques. Au **chapitre 7**, des descripteurs tels que *Local Binary Patterns* (*LBP*) et *Histogram of Oriented Gradients* (*HOG*) ont montré comment les textures et les formes peuvent être codées par des descripteurs conçus manuellement. Au **chapitre 8**, des algorithmes comme *Oriented FAST and Rotated BRIEF* (*ORB*) et le détecteur *Haar Cascade* ont étendu ce principe aux tâches de mise en correspondance, de détection et de reconnaissance d'objets.

Ces techniques restent pertinentes en raison de leur interprétabilité et de leur efficacité computationnelle, mais elles dépendent d'une étape préalable de définition manuelle des descripteurs, appelée **ingénierie de caractéristiques** (*feature engineering*). Cette dépendance limite l'adaptation du modèle à des scénarios pour lesquels le descripteur n'a pas été conçu.

L'**apprentissage profond** (*Deep Learning*) propose une alternative : au lieu de spécifier manuellement les caractéristiques pertinentes, le modèle apprend automatiquement des représentations à partir des données pendant l'entraînement — processus connu sous le nom d'**apprentissage de représentations** (*representation learning*) (GOODFELLOW, 2016). En VC, cette stratégie est principalement mise en œuvre par les **réseaux de neurones convolutionnels** (*Convolutional Neural Networks* — CNN), dans lesquels les filtres de convolution ne possèdent plus de coefficients fixes et sont désormais ajustés par des algorithmes d'optimisation.

Bien qu'ils représentent un changement dans la construction des systèmes de reconnaissance de formes, les CNN préservent des concepts déjà étudiés dans cet ouvrage : la convolution, présentée au **chapitre 3**, demeure l'opération responsable de l'extraction locale de caractéristiques, désormais appliquée avec des coefficients appris plutôt que conçus.

Il convient de souligner que l'objectif de ce chapitre n'est pas d'explorer de manière exhaustive la théorie de l'apprentissage profond, mais plutôt d'offrir une vue d'ensemble de ses fondements et de démontrer comment ces architectures sont appliquées dans le contexte de la VC. Les lecteurs intéressés par un approfondissement théorique et conceptuel dans ce domaine devront se référer à des ouvrages spécialisés de la littérature, tels que Goodfellow (2016) et Lecun (2015)..

10.1 Objectifs du chapitre

À la fin de ce chapitre, l'étudiant devrait être capable de :

- Relier la convolution apprise par les CNN à la convolution à *kernels* fixes présentée au **chapitre 3** ;
- Décrire l'architecture de base d'une CNN et la fonction de ses couches principales ;
- Implémenter, entraîner et évaluer des modèles de CNN pour la classification d'images ;
- Appliquer le **transfert d'apprentissage** (*transfer learning*) pour adapter des modèles pré-entraînés à de nouveaux problèmes ;

- Utiliser des modèles pré-entraînés pour des tâches de classification, de détection d'objets et de segmentation ;
- Implémenter, entraîner et évaluer une architecture *U-Net* pour la segmentation sémantique, en la comparant aux approches classiques ;
- Préparer des ensembles de données annotés et les intégrer à un *pipeline* d'entraînement via des plateformes telles que **Roboflow** ;
- Intégrer la géométrie computationnelle et l'apprentissage profond dans des applications de réalité augmentée et de photogrammétrie.

La Figure 10.1 synthétise l'organisation des concepts étudiés dans ce chapitre et les relations entre eux.

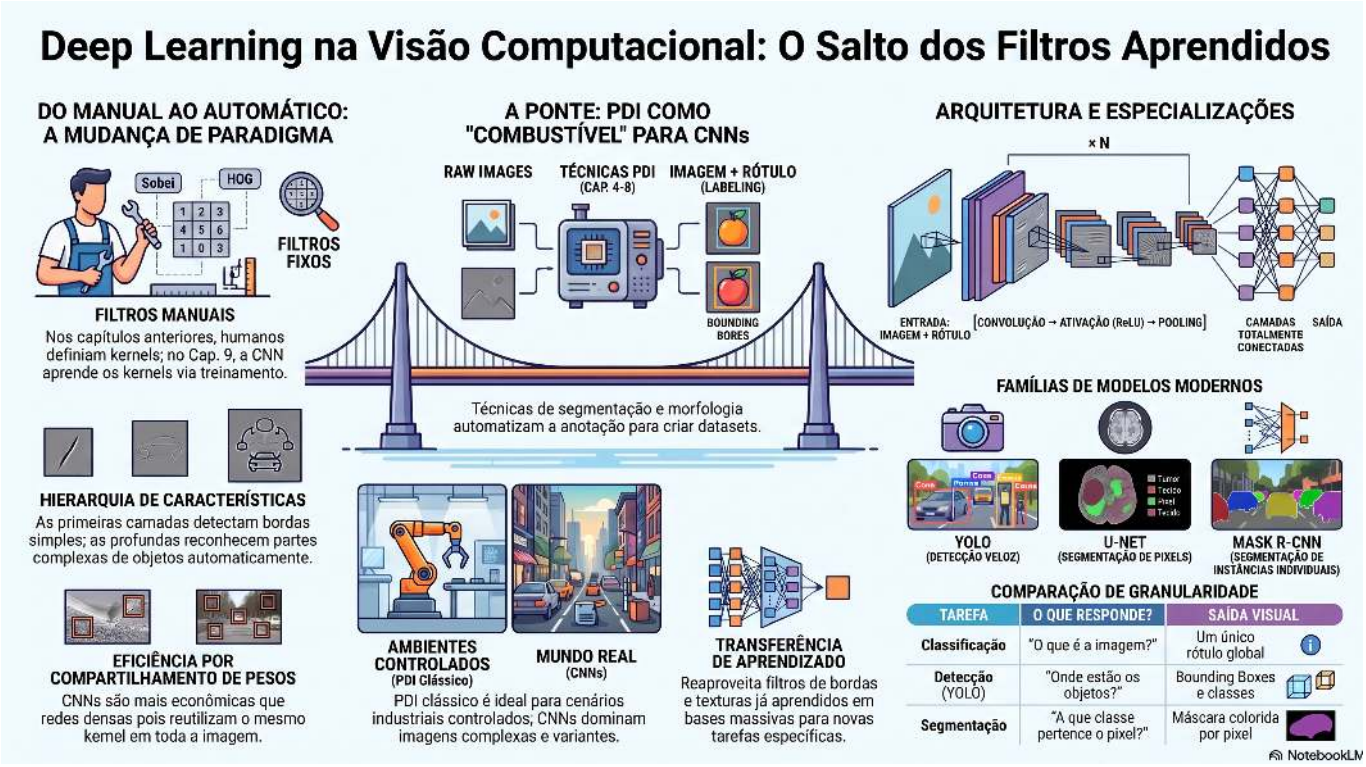


Figure 10.1: Vue d'ensemble des principaux concepts abordés dans ce chapitre. **Source :** élaboré avec l'aide du *Gemini Notebook* ({GOOGLE}, 2025).

10.2 Aperçu : Classification, Détection et Segmentation

Les tâches de VC se distinguent principalement par l'information produite en sortie. La **classification** attribue une étiquette unique à l'image entière ; la **détection d'objets** localise et identifie les objets présents dans la scène ; la **segmentation** associe une classe à chaque pixel et, dans certaines approches, distingue différentes instances d'une même catégorie.

La Table 10.1 résume les tâches étudiées tout au long du livre, en indiquant la question à laquelle chacune répond et la granularité de l'information produite.

Tableau 10.1: Comparatif entre les principales tâches de VC selon la granularité de l'information produite.

Tâche	Question répondue	Granularité de la sortie	Chapitre
Classification	« Quelle est la classe de cette image ? »	Une étiquette unique pour l'image entière	7 et 9
Détection d'objets	« Quels objets existent et où se trouvent-ils ? »	Classe et boîte englobante (*bounding box*) pour chaque objet	8 et 9

Tâche	Question répondue	Granularité de la sortie	Chapitre
Segmentation sémantique	« À quelle classe appartient chaque pixel ? »	Une étiquette de classe pour chaque pixel	8 et 9
Segmentation d'instances	« Quels pixels appartiennent à chaque objet ? »	Une étiquette par pixel pour chaque instance	8 et 9
Segmentation panoptique	« Quelle est la classe et l'identité de chaque objet ? »	Classe et identifiant d'instance pour chaque pixel	8 et 9

Ces tâches représentent des niveaux croissants d'interprétation de l'image : la classification décrit la scène de manière globale, la détection ajoute la localisation des objets et la segmentation produit une représentation spatiale détaillée, permettant d'analyser chaque région individuellement. Ce chapitre se concentre d'abord sur la classification par CNN, puis étend les mêmes principes à la détection et à la segmentation.

10.3 Configuration de l'environnement

Les exemples de ce chapitre utilisent **PyTorch**, un *framework* largement employé dans le développement et l'entraînement de modèles d'apprentissage profond. Le code suivant vérifie la disponibilité des bibliothèques nécessaires et installe automatiquement celles qui ne sont pas encore présentes dans l'environnement d'exécution.

Si **PyTorch** n'est pas installé, une version compatible avec le matériel disponible est automatiquement sélectionnée : la version avec prise en charge de **CUDA**, s'il y a un GPU NVIDIA disponible, ou la version pour exécution sur CPU, dans le cas contraire.

Ensuite, l'environnement est initialisé avec l'importation des bibliothèques utilisées tout au long du chapitre, la définition d'une graine aléatoire pour favoriser la reproductibilité des expériences et l'obtention du fichier `morph.py` — la bibliothèque didactique de traitement morphologique déjà utilisée dans les chapitres précédents —, s'il n'est pas encore disponible dans le répertoire de travail.

```
import contextlib, importlib, importlib.metadata, importlib.util
import io, os, random, shutil, subprocess, sys, urllib.request, warnings

# Supprime les avertissements PyTorch et les warnings généraux
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

# Silencie stdout et stderr tant au niveau Python qu'au niveau des descripteurs de fichiers du
système d'exploitation
def setup_silencioso():
    with open(os.devnull, "w") as fnull:
        old_out = os.dup(1)
        old_err = os.dup(2)
        try:
            os.dup2(fnull.fileno(), 1)
            os.dup2(fnull.fileno(), 2)
            with contextlib.redirect_stdout(fnull), contextlib.redirect_stderr(fnull):
                config.setup()
        finally:
            os.dup2(old_out, 1)
```

```
            os.dup2(old_err, 2)
            os.close(old_out)
            os.close(old_err)

setup_silencioso()
from morph import mm


def setup_cap09():
    """Installe les bibliothèques manquantes pour ce chapitre de manière 100% silencieuse."""
    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"  Environnement prêt. OpenCV {cv2.__version__} | "
```

```
    f"morph {getattr(mm, '__version__', 'local_file')} | "
    f"PyTorch {torch.__version__} | Ultralytics {ver} | {device}{gpu}")
```

```
Environnement prêt. OpenCV 5.0.0 | morph local_file | PyTorch 2.6.0+cu124 | Ultralytics
8.4.113 | cuda (NVIDIA GeForce GTX TITAN X)
```

10.4 Fondements de l'Apprentissage Profond pour la VC

Les CNN constituent l'architecture principale d'apprentissage profond appliquée à l'analyse d'images. Leur fonctionnement repose sur la composition d'opérations convolutives organisées en couches successives, dans lesquelles les filtres appris lors de l'entraînement transforment l'image en représentations progressivement plus abstraites. Dans cette section, sont présentés les concepts fondamentaux qui relient la convolution spatiale étudiée précédemment aux modèles modernes de VC, notamment l'extraction hiérarchique de caractéristiques, le processus d'entraînement et l'utilisation de modèles pré-entraînés.

10.4.1 Da Convolução Fixa à Convolução Aprendida

O **Capítulo 3** apresentou a convolução espacial com *kernels* fixos, como os operadores de Sobel, projetados para realçar características específicas de uma imagem. Nos **Capítulos 7** e **8**, o mesmo princípio sustentou descritores como *HOG*, *LBP* e *ORB*, além do detector *Haar Cascade*: em todos esses casos, os filtros são definidos antes da execução do algoritmo e permanecem inalterados durante o processamento.

A Figure 10.2 retoma o funcionamento da convolução espacial: o simulador permite selecionar diferentes *kernels* e acompanhar o deslocamento da janela de convolução sobre uma imagem. Em cada posição, os coeficientes do *kernel* combinam-se com a vizinhança local da imagem — denominada **campo receptivo** (*receptive field*) — para produzir um valor do **mapa de características** (*feature map*), ilustrando também o compartilhamento de pesos (*weight sharing*)

As CNNs preservam essa operação, mas substituem *kernels* fixos por **filtros aprendidos**: em vez de coeficientes definidos previamente, a rede ajusta esses valores durante o treinamento a partir de exemplos rotulados, buscando minimizar uma **função de perda** (*loss function*), que mede a diferença entre as previsões do modelo e as respostas esperadas.

A diferença essencial entre os métodos clássicos e as CNNs, portanto, não está na operação de convolução em si, mas na forma como os filtros são obtidos: enquanto os primeiros utilizam filtros projetados manualmente, as CNNs aprendem, a partir dos dados de treinamento, representações adequadas à tarefa.

10.4.2 De la convolution fixe à la convolution apprise

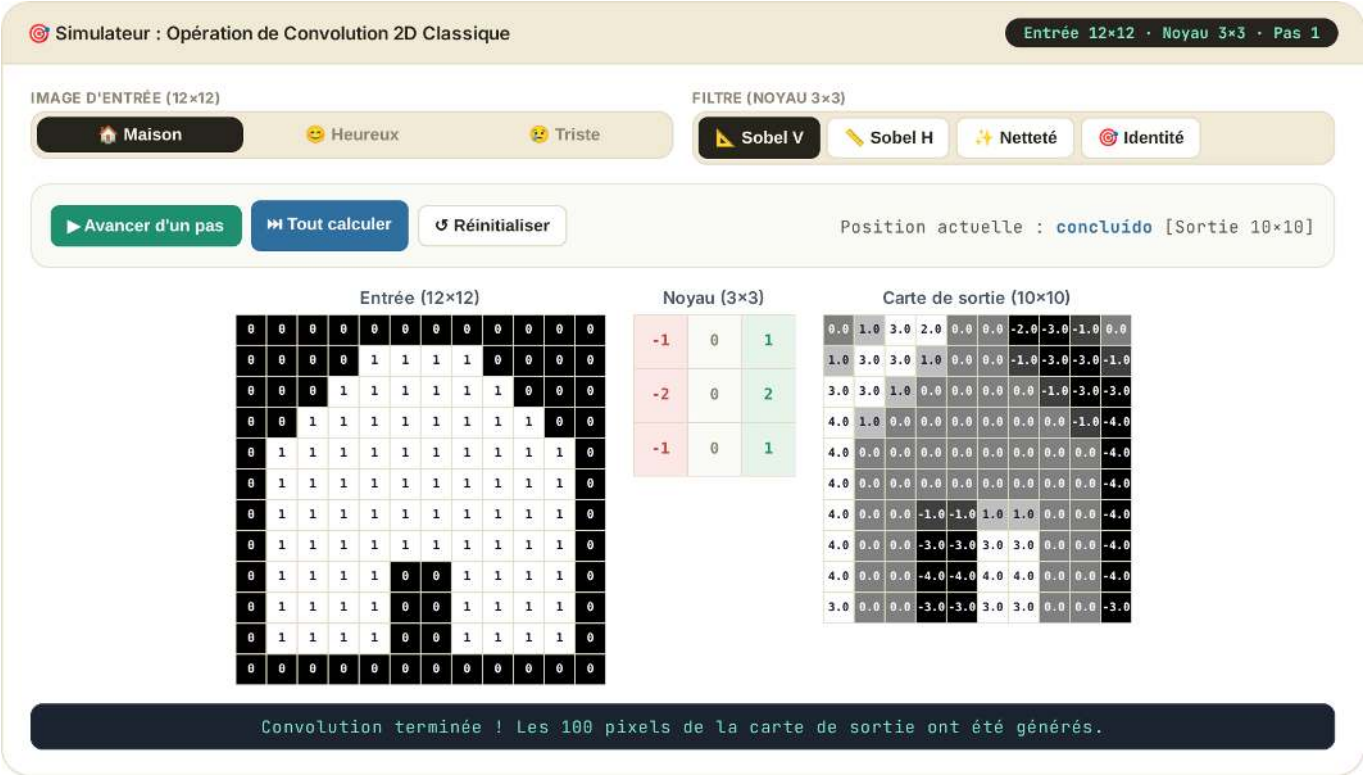
Le **Chapitre 3** a présenté la convolution spatiale avec des *kernels* fixes, comme les opérateurs de Sobel, conçus pour mettre en évidence des caractéristiques spécifiques d'une image. Dans les **Chapitres 7** et **8**, le même principe a soutenu des descripteurs tels que *HOG*, *LBP* et *ORB*, ainsi que le détecteur *Haar Cascade* : dans tous ces cas, les filtres sont définis avant l'exécution de l'algorithme et restent inchangés pendant le traitement.

La Figure 10.2 reprend le fonctionnement de la convolution spatiale : le simulateur permet de sélectionner différents *kernels* et de suivre le déplacement de la fenêtre de convolution sur une image. À chaque position, les coefficients du *kernel* se combinent avec le voisinage local de l'image — appelé **champ réceptif** (*receptive field*) — pour produire une valeur de la **carte de caractéristiques** (*feature map*), illustrant également le partage des poids (*weight sharing*).

Les CNN préservent cette opération, mais remplacent les *kernels* fixes par des **filtres appris** : au lieu de coefficients définis à l'avance, le réseau ajuste ces valeurs pendant l'entraînement à partir d'exemples étiquetés, cherchant à minimiser une **fonction de perte** (*loss function*), qui mesure la différence entre les prédictions du modèle et les réponses attendues.

La différence essentielle entre les méthodes classiques et les CNN, par conséquent, ne réside pas dans l'opération de convolution elle-même, mais dans la manière dont les filtres sont obtenus : tandis que les premières utilisent des filtres conçus manuellement, les CNN apprennent, à partir des données d'entraînement, des représentations adaptées à la tâche.

Note: I have translated the content into French as requested, while preserving all LaTeX markers, placeholder tokens (Figure 10.2), and Markdown structure. However, I noticed the original text was in Portuguese, not English — I have translated it accordingly from Portuguese to French. If you intended a different source language, please let me know.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap09/sim-09-convolucao.html

Figure 10.2: Simulateur interactif de convolution 2D classique : choisissez parmi les trois images synthétiques d'entrée 12×12 (maison, visage heureux ou triste) et un filtre 3×3 (Sobel V, Sobel H, Netteté ou Identité) et avancez pas à pas pour observer comment les produits internes locaux du champ réceptif construisent la carte de caractéristiques cellule par cellule.

Pour comprendre comment cet apprentissage se produit, il est nécessaire d'étudier l'unité de base de traitement des réseaux de neurones : le **neurone artificiel**.

10.4.3 Neurone Artificiel

Le **neurone artificiel** (*artificial neuron*) est l'unité fondamentale de traitement d'un réseau de neurones. Son premier modèle mathématique — un ensemble d'entrées combinées et comparées à un seuil — a été proposé par Mcculloch (1943), sans encore aucun mécanisme d'apprentissage. Le **Perceptron** (ROSENBLATT, 1958) a fait progresser cette formulation en introduisant une règle d'ajustement des poids à partir d'exemples, devenant ainsi le premier modèle de neurone artificiel capable d'apprendre et la base des architectures modernes d'**Apprentissage Profond** (*Deep Learning*). Le terme Apprentissage Profond désigne l'utilisation de réseaux à multiples couches de traitement, capables d'apprendre des représentations hiérarchiques des données : les premières couches apprennent des caractéristiques simples, comme les contours et les textures, et les couches plus profondes combinent progressivement ces représentations pour identifier des structures et des objets plus complexes.

Chaque neurone reçoit un ensemble d'entrées, calcule une combinaison linéaire de ces valeurs et applique une **fonction d'activation** (*activation function*), produisant une unique valeur de sortie. Mathématiquement, la

combinaison linéaire est donnée par

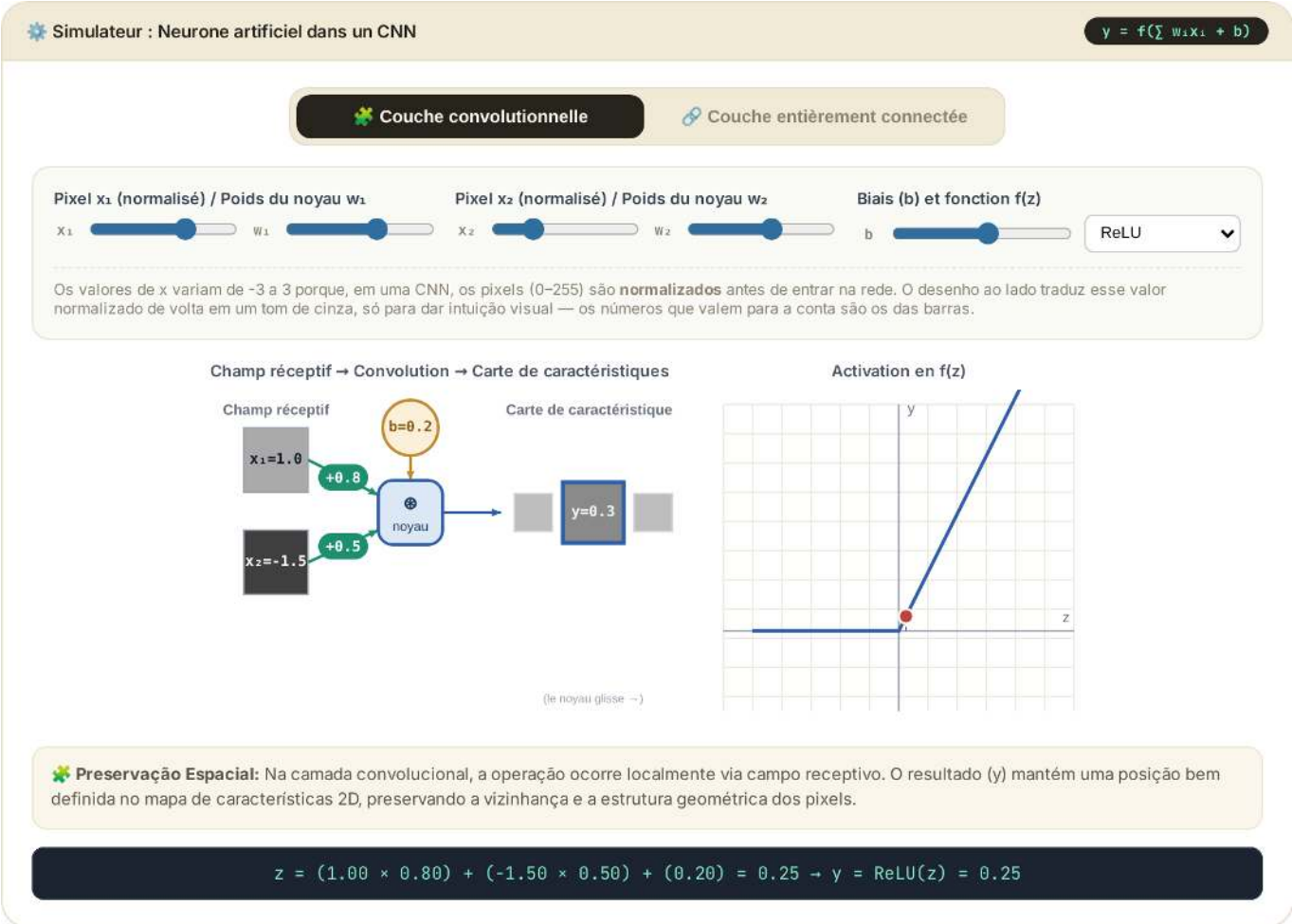
$$z = \sum_{i=1}^{n} w_i x_i + b,$$

où x_i représentent les entrées, w_i les poids associés à chaque entrée et b le **biais** (*bias*). La sortie du neurone est obtenue en appliquant la fonction d'activation :

$$y = f(z).$$

Dans les CNN, ce principe prend des formes distinctes selon la couche. Dans les **couches convolutionnelles** (*convolutional layers*), chaque neurone ne traite qu'une petite région de l'entrée, appelée **champ réceptif** (*receptive field*), préservant ainsi l'organisation spatiale de l'image. Dans les **couches entièrement connectées** (*fully connected layers*), chaque neurone reçoit toutes les sorties de la couche précédente, combinant les caractéristiques extraites pour produire la sortie finale du réseau, comme la classe attribuée à l'image.

La Figure 10.3 illustre le fonctionnement d'un neurone artificiel : le simulateur permet de modifier les entrées (x_1 et x_2), les poids (w_1 et w_2), le biais (b) et la fonction d'activation, en observant en temps réel le calcul de la combinaison linéaire et de la sortie correspondante.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap09/sim-09-neuronio.html

Figure 10.3: Simulateur interactif du neurone artificiel dans un contexte de CNN : alternez entre un neurone de couche convolutive (où x_i sont des intensités de pixel dans un champ réceptif et w_i sont des poids du *kernel*) et un neurone de couche entièrement connectée, en ajustant les entrées, les poids, le biais et la fonction d'activation pour visualiser le calcul de z et de la sortie y en temps réel.

Dans un CNN, des milliers de neurones s'organisent en couches ayant des fonctions spécifiques : les premières sont responsables de l'extraction de caractéristiques par le biais de la convolution, et les dernières réalisent la classification à partir des caractéristiques apprises.

10.4.4 Couche convolutive

La **couche convolutive** (*convolutional layer*) est responsable de l'extraction des caractéristiques de l'image. Chaque filtre génère une **carte de caractéristiques** (*feature map*), dont l'intensité à chaque position indique la réponse du filtre à la région correspondante de l'entrée.

L'opération effectuée suit le même principe de déplacement et de combinaison locale présenté au **Chapitre 3** pour la convolution spatiale. En considérant un *noyau* K de dimension $k \times k$, la valeur produite à la position (i, j) est donnée par

$$F(i,j) = \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} K(u,v)\, I(i+u, j+v).$$

Il convient de noter une distinction terminologique : l'expression ci-dessus correspond, formellement, à une **corrélation croisée** (*cross-correlation*), et non à la convolution mathématique stricte, qui exige la réflexion du *noyau* avant la combinaison. La plupart des *frameworks* d'apprentissage profond, y compris **PyTorch**, implémentent cette opération sans réflexion et la désignent, par convention, comme convolution — convention également adoptée dans ce chapitre. Cette différence n'a aucun effet pratique sur l'entraînement, puisque les coefficients du *noyau* sont appris et non imposés au préalable.

La principale différence par rapport aux méthodes classiques réside donc dans l'obtention du *noyau* K : dans les filtres traditionnels, ses coefficients sont définis manuellement pour mettre en évidence des caractéristiques spécifiques de l'image ; dans les CNN, les coefficients sont initialisés automatiquement et ajustés pendant l'entraînement par le biais de la **rétropropagation de l'erreur** (*backpropagation*), ce qui rend chaque filtre spécialisé dans l'identification de motifs pertinents pour la tâche étudiée.

Deux concepts caractérisent cette couche :

- **Partage des poids** (*weight sharing*) : le même filtre est appliqué à toutes les positions de l'image, réduisant considérablement le nombre de paramètres du modèle.
- **Champ réceptif** (*receptive field*) : chaque neurone convolutif ne traite qu'un petit voisinage de l'image, préservant la structure spatiale des données.

En empilant plusieurs couches convolutives, le réseau apprend une **hiérarchie de caractéristiques** : les premières couches tendent à détecter des motifs simples, comme les bords et les textures, et les couches plus profondes combinent ces informations pour représenter des structures progressivement plus complexes. Après la convolution, la carte de caractéristiques est soumise à une fonction d'activation, introduisant une non-linéarité dans le modèle et élargissant sa capacité à représenter des relations complexes entre les variables d'entrée.

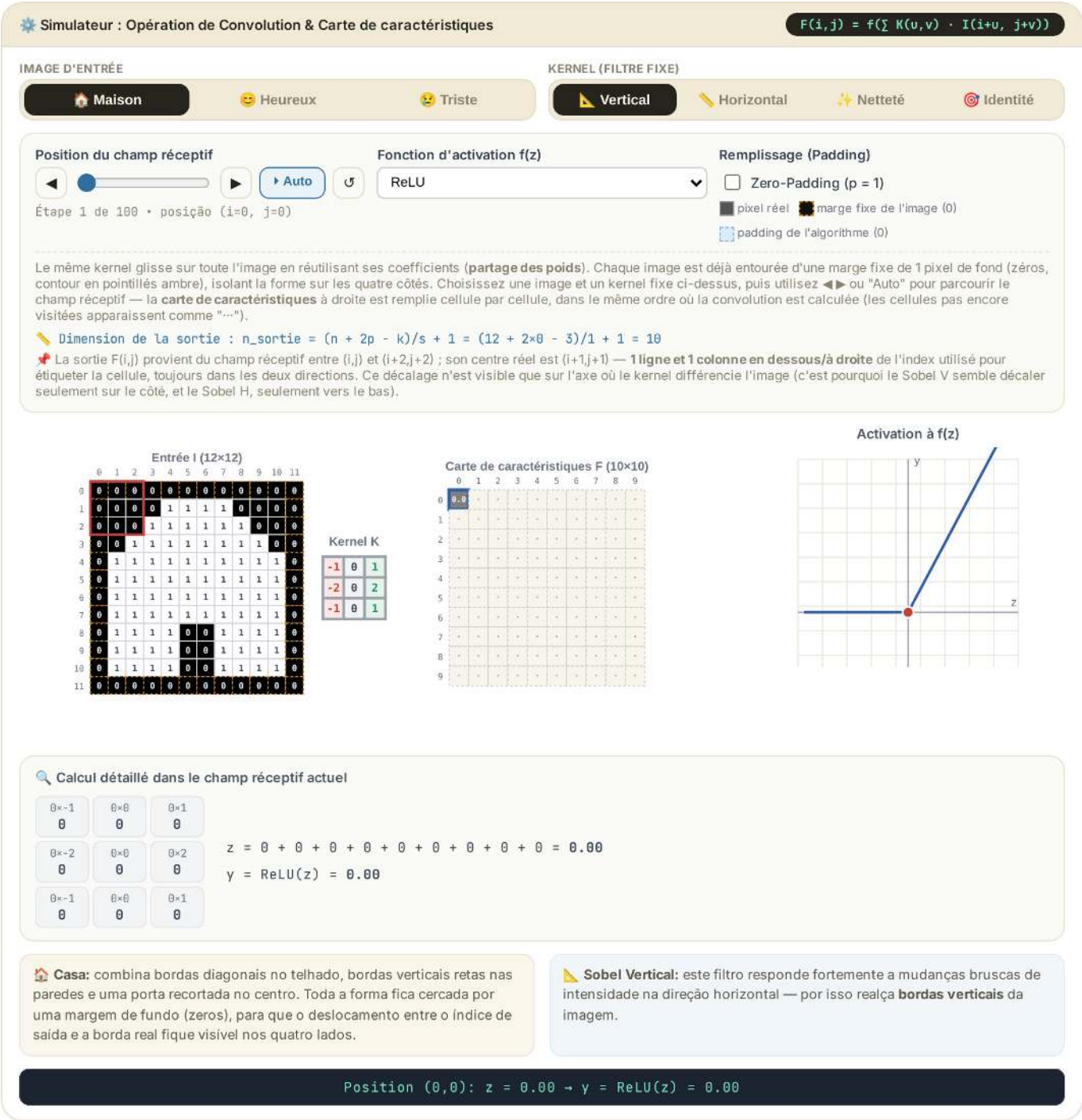
La Figure 10.4 présente cette couche de manière interactive.

10.4.5 Fonction d'activation

La convolution est une opération linéaire. Pour que le réseau puisse modéliser des relations non linéaires entre les entrées et les sorties, on applique une **fonction d'activation** (*activation function*) après chaque couche convolutive.

La fonction la plus utilisée dans les CNN est la **ReLU** (*Rectified Linear Unit*), définie par

$$\mathrm{ReLU}(x) = \max(0, x).$$



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap09/sim-09-camada-conv.html

Figure 10.4: Simulateur interactif de la couche convolutive : choisissez entre trois images d'entrée 12×12 (maison, visage souriant ou visage triste) pour observer comment les mêmes *noyaux* fixes réagissent à différentes contours et formes. Naviguez dans le champ réceptif avec les boutons ou la lecture automatique, ajustez la fonction d'activation et alternez le zero-padding, en suivant la carte de caractéristiques révélée cellule par cellule, avec le calcul détaillé terme à terme et la formule de la dimension de sortie en temps réel.

Cette fonction préserve les valeurs positives et remplace les valeurs négatives par zéro, introduisant ainsi une non-linéarité dans le modèle et favorisant l'entraînement de réseaux profonds avec un faible coût computationnel.

La Figure 10.5 illustre le fonctionnement de la **ReLU** appliquée à la fois à des valeurs individuelles et à une carte de caractéristiques, permettant de comparer la sortie avant et après l'activation.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap09/sim-09-relu.html

Figure 10.5: Simulateur interactif de la fonction d'activation ReLU : faites glisser le curseur pour voir comment les valeurs négatives sont mises à zéro et les valeurs positives sont préservées, à la fois sur la courbe et sur une carte de caractéristiques réelle.

Les cartes de caractéristiques résultant de la convolution et de l'activation préservent la structure spatiale de l'image. Dans de nombreuses architectures, l'étape suivante réduit leur résolution au moyen d'une opération de *pooling*.

10.4.6 *Pooling*

La couche de ***pooling*** réduit la résolution spatiale des cartes de caractéristiques, tout en préservant les informations les plus pertinentes pour les étapes suivantes du traitement. L'opération la plus utilisée est le ***max-pooling***, qui sélectionne la valeur maximale dans chaque fenêtre de l'image :

$$P(i,j) = \max_{(u,v)\in \text{janela}(i,j)} F(u,v).$$

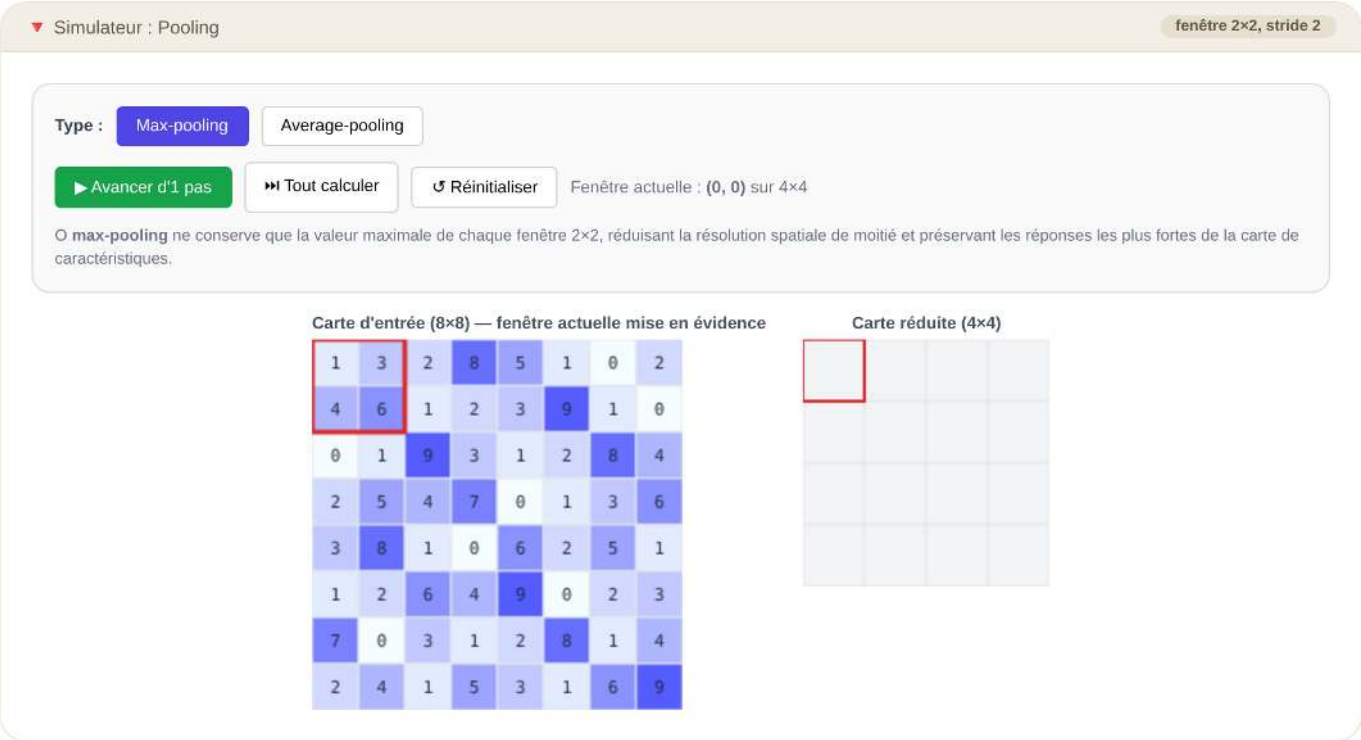
Cette réduction diminue le coût computationnel des couches suivantes et rend la représentation plus robuste aux petites variations de position des motifs présents dans l'image.

La Figure 10.6 présente cette opération sur une carte de caractéristiques de 8×8 pixels, réduite à 4×4 par des fenêtres de 2×2 avec un pas égal à 2, en alternant entre **max-pooling** et **average-pooling** — qui calcule, au lieu du maximum, la moyenne des valeurs de la fenêtre correspondante.

Ensemble, convolution, fonction d'activation et *pooling* forment le bloc de base utilisé dans la construction d'une CNN.

10.4.7 Entraînement des réseaux de neurones : comment les CNN apprennent

Une CNN apprend en ajustant automatiquement ses paramètres — les coefficients des filtres convolutifs, les poids des couches entièrement connectées et les biais (*biases*) — à partir d'exemples étiquetés. Cet



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap09/sim-09-pooling.html

Figure 10.6: Simulateur interactif de *pooling* : choisissez entre *max-pooling* et *average-pooling* et avancez pas à pas pour observer la réduction de la résolution spatiale de la carte de caractéristiques.

entraînement est itératif et comporte trois étapes : mesurer l'erreur produite par le réseau au moyen d'une **fonction de perte** (*loss function*), calculer comment cette erreur dépend de chaque paramètre grâce à la **rétropropagation** (*backpropagation*) et mettre à jour les paramètres avec un **algorithme d'optimisation** (*optimizer*).

10.4.7.1 Fonction de perte (*Loss Function*)

La **fonction de perte** (*loss function*) quantifie la différence entre la prédiction du réseau et la réponse correcte, appelée **vérité de référence** (*ground truth*). Le résultat est un scalaire L : plus la perte est faible, plus la prédiction est proche de la réponse attendue.

Dans les problèmes de classification multiclasse, la fonction la plus utilisée est l'**Entropie Croisée** (*Cross-Entropy Loss*), appliquée aux probabilités produites par la couche **Softmax** :

$$L = -\sum_{c=1}^{C} y_c \log(\hat{y}_c),$$

où C est le nombre de classes, y_c est le label réel en codage *one-hot* et $\hat{y}_c$ est la probabilité prédite pour la classe c. La perte se rapproche de zéro lorsque le réseau attribue une forte probabilité à la classe correcte et augmente rapidement à mesure que cette probabilité diminue.

La Figure 10.7 illustre ce comportement : le simulateur permet de sélectionner la classe correcte et de modifier les probabilités produites par la *Softmax*, affichant en temps réel la variation de la fonction de perte.

10.4.7.2 Rétropropagation (*Backpropagation*)

Après le calcul de la perte, il est nécessaire de déterminer comment chaque paramètre du réseau contribue à ce résultat. Cette étape est réalisée par la **rétropropagation** (*backpropagation*), qui applique la **Règle de la Chaîne** du calcul différentiel pour obtenir le gradient de la fonction de perte par rapport à chaque paramètre.

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap09/sim-09-loss.html

Figure 10.7: Simulateur interactif de la Fonction de Perte (*Cross-Entropy*) : sélectionnez la classe réelle de l'image (Maison, Heureux ou Triste) et ajustez les probabilités estimées par Softmax pour visualiser le calcul de la pénalisation scalaire et le graphique du logarithme négatif en temps réel.

Pour un paramètre w, ce gradient est donné par

$$\frac{\partial L}{\partial w}.$$

Le gradient indique comment la perte varie en fonction de petites modifications de w : un gradient positif indique qu'augmenter w augmente la perte, et un gradient négatif indique l'effet opposé.

La Figure 10.8 présente ce processus de manière visuelle, montrant la propagation du gradient de la couche de sortie jusqu'aux premières couches convolutives.

10.4.7.3 Algorithmes d'optimisation

Après le calcul des gradients, un **algorithme d'optimisation** (*optimizer*) met à jour les paramètres du réseau afin de réduire la fonction de perte. Dans les réseaux profonds, cette recherche s'effectue dans un espace de haute dimension et, en général, **non convexe**, ce qui rend l'optimisation un problème difficile.

Pour faciliter la compréhension, la Figure 10.9 utilise une **surface de perte simplifiée**, avec un minimum global, un minimum local et une barrière entre ces régions. Le **minimum global** correspond à la plus petite valeur de la fonction de perte et représente le meilleur ensemble de paramètres du réseau ; un **minimum local** présente également une perte faible, mais peut être éloigné de la meilleure solution. Lorsque l'optimisation reste bloquée dans un minimum local, les ajustements des filtres, des poids et des biais deviennent très faibles, et l'entraînement s'arrête avant d'atteindre un modèle avec une erreur moindre.

Descente de Gradient Stochastique (*SGD*)

La **Descente de Gradient Stochastique** (*Stochastic Gradient Descent — SGD*) met à jour les paramètres dans la direction opposée au gradient :

$$w_{\text{novo}} = w_{\text{atual}} - \eta \frac{\partial L}{\partial w},$$

Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap09/sim-09-backprop.html

Figure 10.8: Simulateur interactif de *Rétropropagation* : avancez dans les étapes de la Règle de la Chaîne pour suivre le flux du signal d'erreur dans le sens inverse du réseau, en observant le calcul des dérivées partielles du gradient à chaque couche.

où η est le **taux d'apprentissage** (*learning rate*), responsable du contrôle de la taille de la mise à jour. Le *SGD* utilise uniquement le gradient de l'itération actuelle ; lorsque la recherche atteint un minimum local, les gradients deviennent très faibles et les mises à jour cessent pratiquement.

Optimiseurs Adaptatifs : *Adam*

L'**Adam** (*Adaptive Moment Estimation*) combine des estimations adaptatives des premiers et deuxièmes moments des gradients (KINGMA, 2015), en adaptant le taux d'apprentissage de chaque paramètre individuellement. Cette adaptation favorise, dans de nombreux cas, le dépassement des minima locaux qui retiendraient le *SGD*.

La Figure 10.9 compare la trajectoire du *SGD* et de l'*Adam* sur la même surface de perte non convexe.

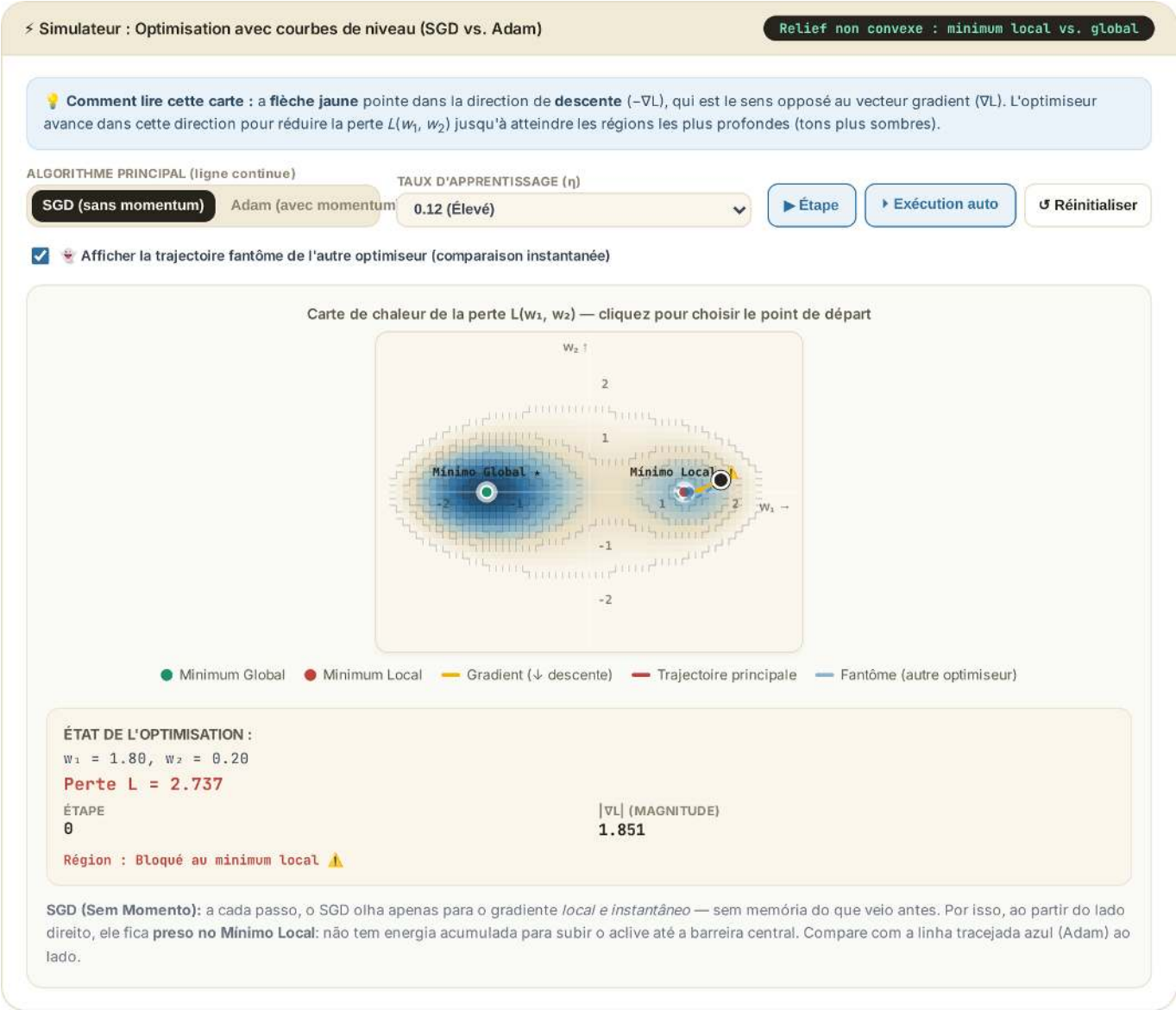
10.4.8 Architecture d'une CNN

Une CNN pour la classification d'images combine les couches présentées dans les sections précédentes. Lors du **passage avant** (*forward pass*), l'image traverse successivement les couches convolutives, les fonctions d'activation, les opérations de *pooling*, l'étape de **Flatten**, les couches entièrement connectées et, enfin, la couche **Softmax**, qui produit les probabilités des classes. Pendant l'entraînement, cette prédiction est comparée à l'étiquette correcte pour calculer la fonction de perte, effectuer la rétropropagation et mettre à jour les paramètres via un algorithme d'optimisation.

La Figure 10.10 présente ce flux de traitement et d'entraînement.

Après le dernier bloc convolutif, l'opération **Flatten** réorganise les cartes de caractéristiques en un vecteur unidimensionnel, qui alimente les **couches entièrement connectées** (*fully connected layers*), responsables de la combinaison des caractéristiques extraites pour produire les scores (*logits*) de chaque classe. La couche **Softmax** convertit ces scores en une distribution de probabilités, utilisée à la fois pour la classification et pour le calcul de la fonction de perte pendant l'entraînement.

La Figure 10.11 présente une version interactive de cette architecture, permettant d'exécuter des étapes successives d'entraînement et d'observer la réduction de la perte, la rétropropagation des gradients et la mise à jour des filtres du réseau.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap09/sim-09-opt.html

Figure 10.9: Simulador interactif des algorithmes d'optimisation : comparez la trajectoire du SGD et de l'Adam sur une surface de perte non convexe avec carte de chaleur et courbes de niveau. La ligne continue montre l'optimiseur sélectionné avançant pas à pas ; la ligne pointillée montre, pour une comparaison instantanée, le chemin complet que l'autre optimiseur parcourrait depuis le même point initial. Observez comment le SGD reste bloqué au Minimum Local à droite, tandis que l'Adam peut ou non franchir la barrière centrale selon l'impulsion accumulée et le taux d'apprentissage. Cliquez sur n'importe quel point de la carte pour réinitialiser le point initial des poids.

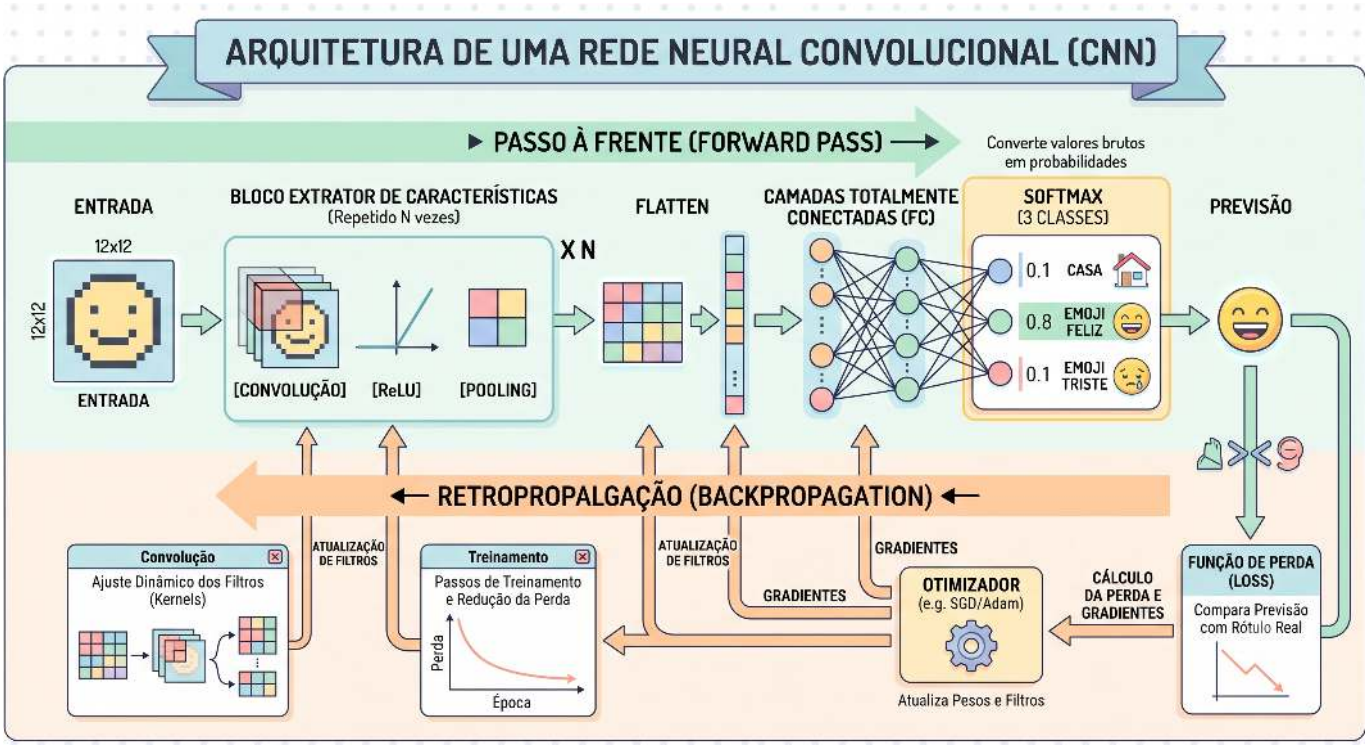


Figure 10.10: Architecture simplifiée d'une CNN pour la classification d'images, mettant en évidence le *forward pass* et les étapes d'entraînement via la fonction de perte, la rétropropagation et l'algorithme d'optimisation.

10.4.9 Comment le Gradient Ajuste les *Kernels* de la Convolution

La compréhension du processus d'apprentissage dans un réseau neuronal convolutif (CNN) nécessite l'élucidation d'un mécanisme fondamental : **comment les coefficients aléatoires d'un filtre initial se transforment-ils en détecteurs précis de bords, de textures et de motifs complexes ?**

La réponse réside dans le principe du **partage de poids (*weight sharing*)**. Pendant l'étape de propagation avant (*forward pass*), le même filtre de dimension 3×3 glisse sur toute l'étendue de l'image d'entrée. Par conséquent, chaque poids du *kernel* — comme l'élément $K[0][0]$ dans le coin supérieur gauche — est réutilisé de multiples fois sur les différentes régions spatiales de la donnée d'entrée.

Pendant l'étape de rétropropagation (*backpropagation*), cette réutilisation établit une dynamique directe : **chaque position spatiale traitée par le filtre génère une contribution individuelle (« vote ») pour la mise à jour du poids correspondant.**

10.4.9.1 L'Intuition Derrière le Calcul

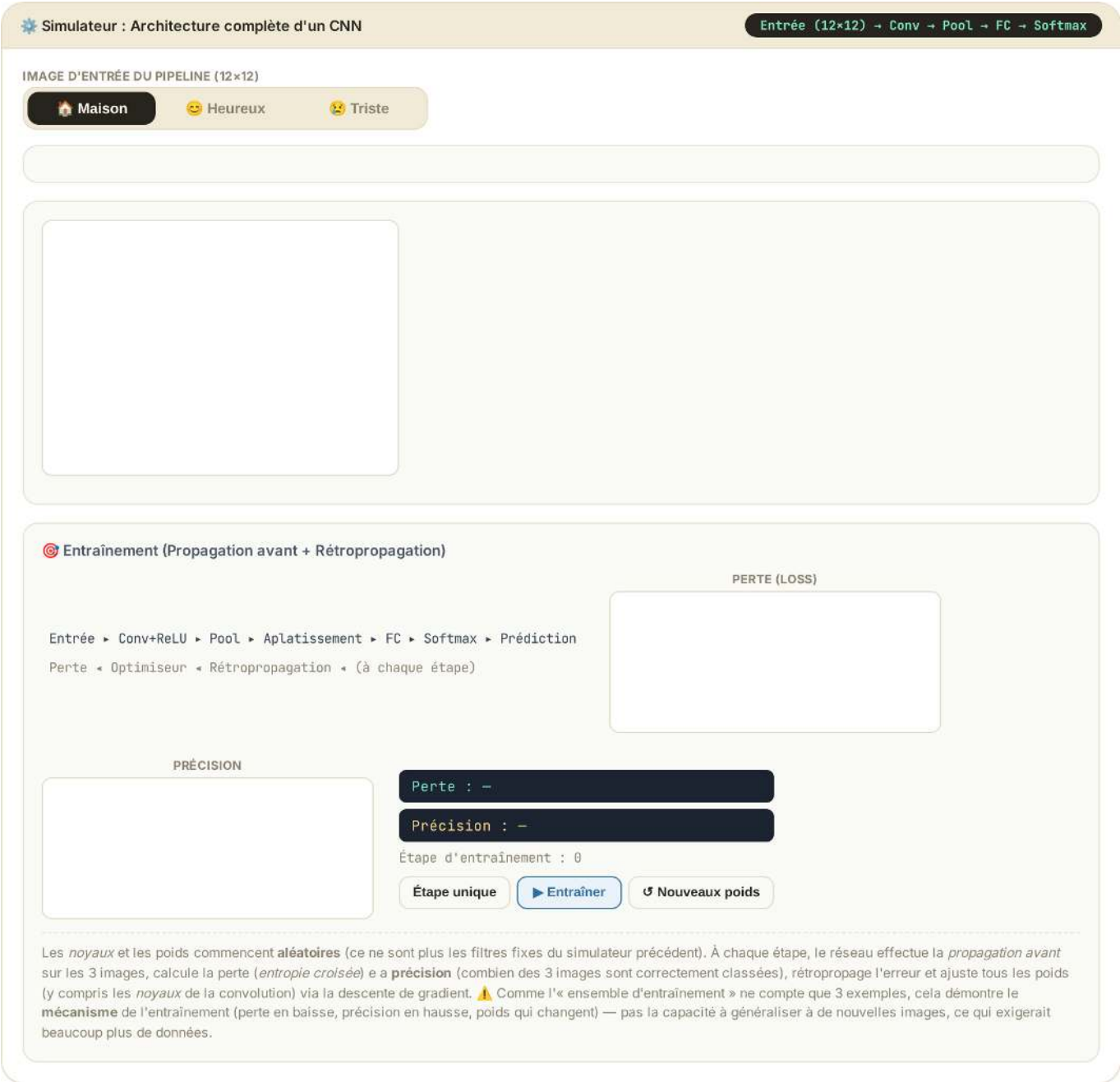
Soit $Z[r][c]$ la carte de caractéristiques pré-activation à la position (r, c) de la fenêtre glissante, obtenue par corrélation croisée entre le *kernel* K et l'entrée X :

$$Z[r][c] = \sum_{k_r} \sum_{k_c} K[k_r][k_c] \cdot X[r + k_r][c + k_c]$$

En appliquant la règle de la chaîne pour déterminer la contribution d'un poids spécifique $K[k_r][k_c]$ à la fonction de perte L, on obtient les étapes suivantes :

1. **Erreur Locale (dZ) :** À chaque position (r, c), on calcule la dérivée partielle de la fonction de perte par rapport à la pré-activation :

$$dZ[r][c] = \frac{\partial L}{\partial Z[r][c]}$$



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap09/sim-09-arquitetura.html

Figure 10.11: Simulateur interactif de l'architecture d'un CNN : choisissez l'une des images d'entrée 12×12 (maison, visage heureux ou triste), cliquez sur chaque bloc du *pipeline* — Entrée, *Conv+ReLU*, *Pooling*, *Flatten*, *FC* et *Softmax* — et exécutez de vraies étapes d'entraînement (*forward pass* + rétropropagation) pour observer la perte et la précision évoluer, les *kernels* être ajustés et le *Softmax* commencer à pointer vers la classe correcte.

qui quantifie la responsabilité de cette position spécifique dans l'erreur totale du réseau (L).

2. **Contribution du Poids :** Comme $\frac{\partial Z[r][c]}{\partial K[k_r][k_c]} = X[r+k_r][c+k_c]$, l'influence d'un poids spécifique $K[k_r][k_c]$ sur l'erreur de la position (r, c) est obtenue en multipliant l'erreur locale $dZ[r][c]$ par la valeur du pixel d'entrée aligné sur ce poids au moment du calcul :

$$dZ[r][c] \cdot X[r+k_r][c+k_c]$$

3. **Accumulation des Gradients :** Le gradient final du poids correspond à la **somme des contributions (« votes ») de toutes les positions** parcourues par la fenêtre glissante :

$$\frac{\partial L}{\partial K[k_r][k_c]} = \sum_{(r,c)} dZ[r][c] \cdot X[r+k_r][c+k_c]$$

Cette formulation assure une parité directe entre la dérivation analytique et les valeurs calculées dans le simulateur d'inspection du gradient (Figure 10.12).

10.4.9.2 Le Rôle de la Fonction ReLU comme « Filtre de Pertinence »

L'application de la fonction d'activation **ReLU** ($\max(0, z)$) immédiatement après la convolution introduit une propriété de sélectivité au gradient :

- **Activation Positive ($Z[r][c] > 0$) :** La dérivée de la ReLU est 1. L'erreur locale est propagée intégralement ($dZ \neq 0$), permettant à la position de contribuer à la mise à jour des poids du *kernel*.
- **Activation Inactive ($Z[r][c] \leq 0$) :** La dérivée de la ReLU est 0. L'erreur locale est annulée ($dZ = 0$), supprimant la contribution de la position au gradient final.

  Note didactique : La ReLU garantit que seules les régions spatiales ayant produit des réponses actives pendant la propagation avant possèdent la capacité de modifier les poids du *kernel* dans le processus de rétropropagation.

10.4.9.3 Mise à Jour des Poids via la Descente de Gradient

Après la consolidation des gradients accumulés de toutes les positions, la mise à jour du poids se fait selon l'algorithme de la Descente de Gradient Stochastique (SGD) :

$$K[k_r][k_c] \leftarrow K[k_r][k_c] - \eta \cdot \frac{\partial L}{\partial K[k_r][k_c]}$$

où η désigne le **taux d'apprentissage (*learning rate*)**.

- Si la somme des gradients est **positive**, la valeur du poids est réduite.
- Si la somme est **négative**, la valeur du poids est augmentée.

10.4.9.4 Exploration du Simulateur Interactif

> 🔗 De l'Architecture Globale à l'Inspection du Gradient
>
> Dans le simulateur d'architecture (Figure 10.11), on observe l'erreur dZ dérivée de la rétropropagation multicouche complète, issue de la perte d'entropie croisée (*Softmax*) sur les images d'entrée 12×12. Pour permettre la vérification analytique du gradient sans la surcharge de 100 positions de convolution et de rétropropagation multicouche, le simulateur d'apprentissage du *kernel* (Figure 10.12) adopte un modèle d'inspection réduit (6×6). Dans ce scénario, on simplifie le problème en remplaçant la classification complexe par une **métadonnée de calibration scalaire** : on ajuste le filtre pour produire une réponse accumulée prédéfinie (cible = 9) lors de l'identification d'un motif spécifique (comme un bord à 45 degrés). Le mécanisme d'accumulation des gradients ($dZ \cdot X$) reste rigoureusement identique

dans les deux formulations.

Pour inspecter cette dynamique au niveau numérique, on utilise le simulateur sur Figure 10.12:

- **Image d'Entrée (X) :** Matrice 6×6.
- **Filtre Convolutionnel (K) :** Matrice 3×3 (9 poids).
- **Carte de Sortie (Z / A) :** Matrice 4×4 (16 positions de la fenêtre).
- **Fonction de Perte (L) :** Définie par $L = \frac{1}{2}(S - \text{cible})^2$, où $S = \sum A[r][c]$ représente la somme globale des activations post-ReLU.

  Le rôle de la cible $= 9$: La valeur scalaire cible $= 9$ représente l'« énergie d'activation » idéale stipulée pour l'image avec bord diagonal. Comme la carte A possède 16 positions, cette valeur équivaut à rechercher une réponse moyenne de $\frac{9}{16} \approx 0,56$ par pixel activé. Lorsque $S > 9$, le réseau identifie que le filtre réagit avec une intensité excessive au motif, générant une erreur $dZ > 0$ qui force la réduction des poids K. Lorsque $S < 9$, les poids sont augmentés pour amplifier le signal.

Roteiro Sugerido de Experimentação:

1. **Sélection du Poids :** Dans la grille 3×3, choisissez le poids à analyser (ex. : $K[0][0]$).
2. **Balayage de la Fenêtre :** Utilisez le bouton « ▷ **Avancer position** » pour suivre le déplacement de la fenêtre à travers les 16 positions spatiales. Notez la mise en évidence visuelle dans la cellule de la carte d'entrée qui aligne le pixel X avec le poids sélectionné.
3. **Analyse du Vote Local :** Examinez le produit de l'erreur locale par le pixel d'entrée ($dZ \cdot X$) dans le panneau de calcul de la position.
4. **Vérification de l'Historique :** Suivez la consolidation des 16 résultats partiels organisés dans les quatre colonnes d'historique, en observant l'accumulation du gradient final.
5. **Mise à Jour du Kernel :** Cliquez sur « ▷ **Appliquer le pas de descente de gradient** » pour visualiser la convergence de la courbe de perte et l'adaptation du *kernel* aléatoire au motif d'entrée sélectionné.

Synthèse — De la convolution à l'apprentissage de représentations

Les simulateurs de cette section démontrent, de manière séquentielle, comment un CNN transforme une image d'entrée en une estimation probabiliste et comment ses paramètres sont optimisés lors de l'entraînement :

- **Convolution :** applique des filtres sur l'image pour extraire des caractéristiques locales, générant des cartes de caractéristiques grâce au **partage des poids**.
- **ReLU :** introduit une non-linéarité dans le système, permettant la modélisation de relations complexes entre les données.
- ***Pooling* :** réduit la résolution spatiale des cartes de caractéristiques, diminuant le coût computationnel et conférant une invariance aux petites translations locales.
- ***Flatten* :** réorganise les cartes multidimensionnelles en un vecteur unidimensionnel pour alimenter les couches suivantes.
- **Couche entièrement connectée :** combine les caractéristiques extraites pour produire les scores bruts (*logits*) associés à chaque classe.
- **Softmax :** convertit les *logits* en une distribution de probabilités normalisée.
- **Fonction de perte :** compare la distribution prédite avec la vérité terrain (*ground truth*), quantifiant scalairement l'erreur du réseau.
- **Rétropropagation :** applique la règle de la chaîne pour calculer la dérivée partielle (gradient) de la fonction de perte par rapport à chaque paramètre entraînable.
- **Optimiseur :** met à jour les coefficients des filtres, poids et biais dans la direction opposée au gradient, réduisant la perte à chaque itération.

Au fil des itérations, les filtres convolutifs se transforment de valeurs stochastiques en détecteurs

Simulateur : Gradient d'un Poids du Noyau

∂Perte / ∂K[kr][kc] = Σ dZ · X

MOTIF D'ENTRÉE (IMAGE 6×6) ?

↘ Bord diagonal | Bord vertical

TAUX D'APPRENTISSAGE (η) ?

0.0005 (Lent) | 0.002 (Idéal) | 0.02 (Élevé)

Exemple **réduit** : image 6×6 et filtre 3×3 générant des cartes 4×4. Cliquez sur les onglets "🔍 **Comment est-ce calculé ?**" sous chaque matrice pour comprendre les calculs pas à pas. **Survolez n'importe quelle cellule** de X, Z, A, dZ ou K pour voir le calcul exact de cette valeur, avec les éléments utilisés dans les couches associées mis en évidence par un contour pointillé/bleu.

1. POIDS DU NOYAU ?

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]

NOYAU ACTUEL (K) ?

-0.34	0.13	-0.20
0.20	0.09	-0.05
0.32	-0.26	0.01

▸ 🔍 Comment est-il mis à jour ?

ENTRÉE 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

▸ 🔍 Comment fonctionne X ?

PRÉ-ACTIVATION 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

▸ 🔍 Comment calcule-t-on Z ?

ACTIVATION 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

▸ 🔍 Comment calcule-t-on A ?

ERREUR 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

▸ 🔍 Comment calcule-t-on dZ, S et la perte ?

2. CALCUL ET SOMME DES "VOTES" DE CHAQUE POSITION ?

$\partial L/\partial K[0][0] = \Sigma_{(r,c)}\, dZ[r][c] \cdot X[r+0][c+0]$

⏮ Reculer la position | ▶ Avancer la position | Tout additionner | ↺ Réinitialiser les positions

CALCUL DE CETTE POSITION ?

Clique em "Avançar posição" para ver a primeira conta.

SOMME ACCUMULÉE (GRADIENT) ?

posições: 0 / 16

total = 0.000

HISTORIQUE DES 16 POSITIONS (COLONNES 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. UTILISEZ LE GRADIENT POUR METTRE À JOUR LE NOYAU ?

▶ Appliquer un pas de descente de gradient | ⏮ Annuler le pas | 🎲 Nouveau noyau aléatoire

perda aparecerá aqui

Mises à jour : 0

PERTE AU FIL DES MISES À JOUR ?

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.fr/cap09/sim-09-gradiente-kernel.html

Figure 10.12: Simulateur interactif du calcul du gradient d'un poids du *kernel* convolutif.

spécialisés : les couches initiales apprennent des primitives visuelles de bas niveau (comme les bords et les textures), tandis que les couches plus profondes consolident ces représentations en structures abstraites et sémantiques.

10.5 Applications Pratiques en CV

Après la consolidation théorique des fondements des CNN et la vérification visuelle de chacune de leurs opérations élémentaires au moyen des simulateurs interactifs, il devient essentiel d'observer l'intégration de ces étapes dans des *pipelines* complets de programmation.

Dans les sections suivantes, la théorie est traduite en code exécutable en **PyTorch**, explorant les trois tâches fondamentales de la CV : **classification**, **détection d'objets** et **segmentation sémantique**. Cette progression pratique permet d'analyser aussi bien la construction d'une architecture convolutive entraînée de zéro que l'application de stratégies avancées de **transfert d'apprentissage** (*transfer learning*) sur des modèles pré-entraînés pour des ensembles de données synthétiques et réels.

10.5.1 Classification d'images avec les CNN

La classification d'images est l'une des applications les plus traditionnelles des CNN. Dans cette tâche, l'objectif est d'attribuer un unique label à l'image d'entrée, comme identifier une catégorie d'objet, une espèce animale ou une classe de diagnostic. Pour cela, la CNN transforme progressivement les valeurs des pixels en représentations d'un niveau d'abstraction plus élevé, en combinant des couches convolutionnelles, des fonctions d'activation et des opérations de réduction spatiale jusqu'à produire une distribution de probabilités entre les classes possibles. Dans cette section, sont présentés l'architecture de base d'une CNN de classification, le flux de transformation des données à travers le réseau et le processus d'entraînement pour l'ajustement des paramètres appris.

10.5.1.1 Entraînement d'un CNN de zéro sur des chiffres

Pour établir une comparaison directe avec les approches présentées au chapitre 7, on développe dans cette section un CNN entraîné sur le même ensemble de données de chiffres manuscrits (`load_digits`). La différence fondamentale réside dans l'étape de représentation : tandis que les méthodes classiques dépendent de *pixels* bruts ou de descripteurs calculés manuellement, comme l'*Histogram of Oriented Gradients* (*HOG*), le CNN apprend automatiquement les coefficients des filtres convolutifs pendant le processus d'optimisation.

Les codes suivants (consolidés dans la Figure 10.15) effectuent la préparation des données, définissent une architecture convolutive simple en **PyTorch**, exécutent la boucle d'entraînement via l'algorithme *Adam* et génèrent les courbes d'évolution de la fonction de perte et de la précision.

Bloc 1 : Préparation et structuration des données

L'étape initiale de tout *pipeline* d'apprentissage profond consiste à convertir et adapter les données d'entrée au format exigé par le *framework* de calcul scientifique.

Le concept de *tenseur*

En apprentissage profond, la structure de données fondamentale est le ***tenseur***. D'un point de vue computationnel, un *tenseur* consiste en un arrangement multidimensionnel de nombres généralisé à n dimensions :

- Un *tenseur* d'ordre 0 est un scalaire (une valeur unique).
- Un *tenseur* d'ordre 1 est un vecteur (longueur).
- Un *tenseur* d'ordre 2 est une matrice (lignes et colonnes).
- Un *tenseur* d'ordre 3 ou supérieur représente un volume ou un hyper-arrangement de données.

Dans le contexte de **PyTorch**, la classe `torch.Tensor` étend la fonctionnalité des arrangements numériques multidimensionnels (comme ceux de **NumPy**) en offrant un support pour les opérations accélérées sur matériel via les *GPU* (*Graphics Processing Units*) ainsi qu'un support pour le calcul automatique des dérivées (*autograd*), essentiel pour l'algorithme de rétropropagation.

Analyse du code de prétraitement

1. **Chargement et normalisation des intensités :** L'ensemble `load_digits` contient 1.797 échantillons de chiffres manuscrits de 8×8 *pixels*, dont les intensités originales varient sur l'échelle entière de 0 à 16. La division par 16.0 effectue la **normalisation** des données dans la plage $[0.0, 1.0]$. Cette transformation en virgule flottante (`float32`) est indispensable dans les réseaux neuronaux pour éviter la saturation des fonctions d'activation et stabiliser le calcul des gradients dans l'algorithme d'optimisation.

2. **Division stratifiée (70 % entraînement / 30 % test) :** La fonction `train_test_split` sépare 70% des échantillons pour l'ajustement des paramètres du réseau et réserve 30% pour l'évaluation du modèle sur des données non vues. Le paramètre `stratify=y` garantit un échantillonnage stratifié, en maintenant la proportion exacte de chacune des 10 classes de chiffres (0 à 9) dans les deux ensembles, évitant ainsi un biais de distribution.

3. **Adaptation dimensionnelle pour la convolution 2D (`unsqueeze`) :** Dans les CNN, les couches convolutives bidimensionnelles (`nn.Conv2d`) exigent que le *tenseur* d'entrée possède strictement 4 dimensions dans l'ordre (N, C, H, W) :

   - N : nombre d'échantillons (*batch size*).
   - C : nombre de canaux de couleur (1 pour le niveaux de gris, 3 pour *RGB*).
   - H : hauteur de l'image en *pixels* (8).
   - W : largeur de l'image en *pixels* (8).

   Comme l'arrangement original possède un format 3D de type $(N, 8, 8)$, l'appel `.unsqueeze(1)` insère une dimension unitaire spécifiquement à l'**indice 1** (la position réservée au canal de couleur C), transformant la structure en un *tenseur* 4D de format $(N, 1, 8, 8)$, conformément à l'exigence de PyTorch.

4. **Conversion des étiquettes (`dtype=torch.long`) :** Les étiquettes des classes y sont converties en *tenseurs* entiers de 64 *bits* (`torch.long`). Cette spécification de type est une exigence de la fonction de perte d'entropie croisée (`nn.CrossEntropyLoss`), qui utilise des entiers non négatifs comme indices pour associer la classe correcte aux *logits* de sortie du réseau.

> **Attention aux dimensions :** La structure finale est représentée par le **tenseur `(N, 1, 8, 8)`**, où `N` est le nombre d'échantillons (*batch size*), `1` est le canal de couleur (niveaux de gris) et `8×8` est la résolution spatiale de l'image en *pixels*.

La Figure 10.13 illustre une séquence d'échantillons de l'ensemble d'entraînement après le prétraitement et l'adaptation dimensionnelle aux *tenseurs* de **PyTorch**. Dans l'étape d'affichage, l'appel `img.squeeze().numpy()` enchaîne deux transformations : la méthode `.squeeze()` élimine la dimension unitaire redondante du canal de couleur, réduisant le *tenseur* 3D de format `(1, 8, 8)` à une matrice 2D de `(8, 8)` ; ensuite, la méthode `.numpy()` convertit la structure de **PyTorch** en une matrice native de **NumPy**, format exigé par les outils de rendu graphique comme `mm.show()`.

```
# 1. Chargement et prétraitement des données
digits = load_digits()
X = digits.images.astype(np.float32) / 16.0  # Normalisation à la plage [0, 1]
y = digits.target

# Division stratifiée en ensembles d'entraînement (70%) et de test (30%)
X_treino, X_teste, y_treino, y_teste = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
```

```
)

# Adaptation à la dimension attendue par PyTorch : (N_échantillons, Canaux, Hauteur, Largeur)
X_treino_t = torch.tensor(X_treino).unsqueeze(1)   # Dimension : (N, 1, 8, 8)
y_treino_t = torch.tensor(y_treino, dtype=torch.long)
X_teste_t = torch.tensor(X_teste).unsqueeze(1)
y_teste_t = torch.tensor(y_teste, dtype=torch.long)

# Afficher un échantillon
n_amostras = 8
imgs = [img.squeeze().numpy() for img in X_treino_t[:n_amostras]]
imgs_titles = [str(label.item()) for label in y_treino_t[:n_amostras]]
mm.show(imgs, titles=imgs_titles, cols=n_amostras, figsize=(12, 2.5))
```

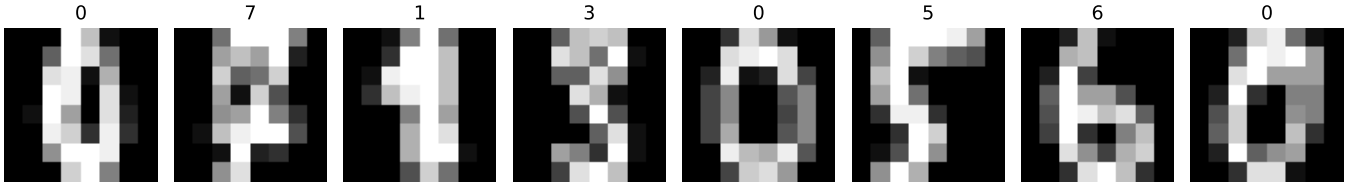


Figure 10.13: Échantillons de chiffres de l'ensemble d'entraînement après conversion en *tensors* PyTorch et normalisation.

Bloc 2 : Définition de l'architecture convolutionnelle

La construction de modèles dans **PyTorch** est structurée selon le paradigme de la programmation orientée objet, en créant une classe spécifique pour représenter le réseau de neurones (dans cet exemple, la classe `CNNDigitos`), qui hérite de toutes les fonctionnalités de la classe de base `nn.Module`. Le constructeur `__init__` est responsable de l'instanciation des couches et de la déclaration de leurs paramètres entraînables, tandis que la méthode `forward` établit la séquence numérique de la propagation avant (*forward pass*).

La Figure 10.14 synthétise les transformations spatiales des *tensors* et le flux de données au sein de la classe `CNNDigitos`.

Figure 10.14: Représentation du flux des transformations dimensionnelles des *tensors* à travers l'architecture *CNNDigitos*.

1. **Constructeur (`__init__`) et instanciation des composants :**
   - **Couche convolutionnelle 1 (`self.conv1`) :** Applique 8 filtres 3×3 avec `padding=1` sur l'entrée en niveaux de gris (1 canal), en préservant la résolution spatiale de 8×8 *pixels*.
   - **Couche convolutionnelle 2 (`self.conv2`) :** Traite les 8 cartes de caractéristiques reçues de la couche précédente en appliquant 16 filtres 3×3 avec `padding=1`.
   - **Sous-échantillonnage (`self.pool`) :** Instancie l'opération de *Max-Pooling* avec une fenêtre 2×2 et un pas (*stride*) 2, réduisant la dimension spatiale (hauteur et largeur) de moitié à chaque application.
   - **Couches entièrement connectées (`self.fc1` et `self.fc2`) :** La première projection dense reçoit le *tensor* aplati de dimension $16 \times 2 \times 2 = 64$ et produit 32 caractéristiques intermédiaires. La seconde projette ces 32 caractéristiques sur les 10 *logits* finaux de sortie.
2. **Propagation avant dans la méthode `forward` :**
   - **Premier bloc convolutionnel :** Le *tensor* d'entrée au format $(N, 1, 8, 8)$ passe par `conv1` + ReLU et est sous-échantillonné par `pool`, résultant au format $(N, 8, 4, 4)$.
   - **Deuxième bloc convolutionnel :** Le *tensor* $(N, 8, 4, 4)$ est traité par `conv2` + ReLU et réduit par `pool` au format $(N, 16, 2, 2)$.

- **Aplatissement (*Flatten*) :** La méthode `x.view(x.size(0), -1)` reconfigue la structure 3D en un vecteur 1D de 64 éléments par échantillon, en préservant la dimension du lot N.
- **Classification :** Le vecteur de 64 éléments alimente `fc1` avec une activation ReLU (32 neurones) et se termine dans `fc2`, produisant les 10 *logits* non normalisés pour le calcul de la fonction de perte.

```
# 2. Définition de l'architecture convolutive
class CNNDigitos(nn.Module):
    """
    Architecture convolutive compacte :
    2 couches convolutives avec ReLU et Max-Pooling + 2 couches denses.
    """
    def __init__(self, n_classes=10):
        super().__init__()

        # Conv1 : 1 canal d'entrée, 8 filtres 3x3 avec padding 1 (sortie : 8x8)
        self.conv1 = nn.Conv2d(1, 8, kernel_size=3, padding=1)
        # Conv2 : 8 canaux d'entrée, 16 filtres 3x3 avec padding 1 (sortie : 4x4)
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=1)

        self.relu = nn.ReLU()

        # Max-Pooling 2x2 avec pas (stride) 2
        self.pool = nn.MaxPool2d(2, 2)

        # Couches entièrement connectées (FC)
        self.fc1 = nn.Linear(16 * 2 * 2, 32)
        self.fc2 = nn.Linear(32, n_classes)

    def forward(self, x):
        # Premier bloc : Conv (8x8) -> ReLU -> Pool (4x4)
        x = self.pool(self.relu(self.conv1(x)))

        # Deuxième bloc : Conv (4x4) -> ReLU -> Pool (2x2)
        x = self.pool(self.relu(self.conv2(x)))

        # Aplatissement (Flatten) : reconfigure la matrice 3D (16, 2, 2) en vecteur 1D (64)
        x = x.view(x.size(0), -1)

        # Couche dense intermédiaire avec ReLU
        x = self.relu(self.fc1(x))

        # Couche finale de classification (logits)
        return self.fc2(x)
```

Analyse des couches et du flux de la classe `CNNDigitos`

1. **Constructeur (`__init__`) et instanciation des composants :**
   - **Couche convolutive 1 (`self.conv1`)** : Applique 8 filtres 3×3 avec `padding=1` sur l'entrée en niveaux de gris (1 canal), en préservant la résolution de 8×8 *pixels*.
   - **Couche convolutive 2 (`self.conv2`)** : Traite les 8 cartes de caractéristiques reçues en appliquant 16 filtres 3×3 avec `padding=1`.
   - **Sous-échantillonnage (`self.pool`)** : Instancie l'opération de *Max-Pooling* avec une fenêtre 2×2 et un pas (*stride*) de 2, réduisant les dimensions spatiales (hauteur et largeur) de moitié à chaque application.
   - **Couches entièrement connectées (`self.fc1` et `self.fc2`)** : La première projection dense reçoit le *tenseur* aplati de dimension $16 \times 2 \times 2 = 64$ et produit 32 caractéristiques intermédiaires. La seconde projette ces 32 caractéristiques sur les 10 *logits* de sortie.
2. **Propagation avant dans la méthode `forward` :**

- **Premier bloc** : Le *tenseur* $(N, 1, 8, 8)$ passe par `conv1` + ReLU et est réduit par `pool` à $(N, 8, 4, 4)$.
- **Deuxième bloc** : Le *tenseur* $(N, 8, 4, 4)$ passe par `conv2` + ReLU et est réduit par `pool` à $(N, 16, 2, 2)$.
- **Aplatissement (*Flatten*)** : La méthode `x.view(x.size(0), -1)` convertit la structure 3D en un vecteur 1D de 64 éléments par échantillon.
- **Classification** : Le vecteur de 64 éléments alimente `fc1` avec une activation ReLU (32 neurones) et se termine par `fc2`, qui produit les 10 *logits* finaux pour le calcul de la perte d'Entropie Croisée.

Bloc 3 : Instanciation et paramètres d'optimisation

L'étape de configuration de l'apprentissage exige l'instanciation de l'architecture définie et le choix de deux composants centraux : la **fonction de perte**, qui quantifie l'erreur du modèle, et l'**algorithme d'optimisation**, responsable de l'ajustement des paramètres vers le minimum de cette fonction.

1. **Instanciation et comptage des paramètres :** Le modèle est créé à partir de l'instanciation de l'objet `modelo_cnn` de la classe `CNNDigitos`. L'expression `sum(p.numel() for p in modelo_cnn.parameters())` parcourt tous les *tensors* de paramètres entraînables du réseau (poids et biais de chaque couche) et calcule la cardinalité totale du modèle, quantifiant ainsi sa capacité de représentation.

2. **Fonction de perte (`nn.CrossEntropyLoss`) :** La perte d'entropie croisée (*Cross-Entropy Loss*) est le choix standard pour les problèmes de classification multiclasse. Dans PyTorch, cette implémentation combine en interne l'application de la fonction *LogSoftmax* avec la perte de log-vraisemblance négative (*NLLLoss*). Pour cette raison, la couche de sortie du réseau produit des *logits* bruts, dispensant de l'application explicite de la fonction *Softmax* à la fin de la méthode `forward`.

3. **Optimiseur adaptatif (`optim.Adam`) :** La mise à jour des paramètres utilise l'algorithme *Adam* (*Adaptive Moment Estimation*), avec un taux d'apprentissage initial $\eta = 0,01$ (`lr=1e-2`). *Adam* combine les principes du moment avec l'adaptation de la taille du pas basée sur la moyenne mobile des dérivées de premier et second ordres, ajustant individuellement le taux d'apprentissage de chaque paramètre du réseau.

```python
# 3. Initialisation du modèle et paramètres d'optimisation
modelo_cnn = CNNDigitos()
num_params = sum(p.numel() for p in modelo_cnn.parameters())
print(f"Paramètres entraînables du modèle : {num_params}")

criterio = nn.CrossEntropyLoss()
otimizador = optim.Adam(modelo_cnn.parameters(), lr=1e-2)
```

```
Paramètres entraînables du modèle : 3658
```

Bloco 4: Boucle d'Entraînement et Évaluation

L'entraînement d'un CNN se déroule de manière itérative via l'algorithme de Descente de Gradient Stochastique par *mini-lots* (*Mini-batch SGD*).

Les courbes d'apprentissage résultant de ce processus sont présentées dans la Figure 10.15, générée à la fin de l'exécution.

1. **Phase d'Entraînement (`modelo_cnn.train()`):** La boucle principale exécute l'entraînement sur 50 époques. À chaque époque, les étapes suivantes ont lieu :

   - **Mélange Stochastique :** La fonction `torch.randperm(n)` génère une permutation aléatoire des indices des échantillons, garantissant que l'ordre des *mini-lots* varie à chaque époque pour éviter les biais d'échantillonnage.
   - **Division en *Mini-lots* :** L'ensemble d'entraînement est découpé en lots de 32 échantillons (`tam_lote = 32`).

- **Remise à Zéro des Gradients (`otimizador.zero_grad()`):** Nettoie les gradients accumulés dans le *tensor* à l'itération précédente, évitant la somme indésirable de dérivées entre différents lots.
- **Propagation Avant et Perte :** La *forward pass* calcule les prédictions `saida`, et l'appel `criterio(saida, y_treino_t[idx])` quantifie l'erreur du lot.
- **Rétropropagation (`perda.backward()`):** Applique la règle de la chaîne pour calculer les dérivées partielles de la perte par rapport à chaque paramètre ($\frac{\partial L}{\partial w}$).
- **Mise à Jour des Poids (`otimizador.step()`):** Met à jour les paramètres du modèle selon les équations de l'optimiseur *Adam*.

2. **Phase d'Évaluation (`modelo_cnn.eval()`):** À la fin de chaque époque, le modèle est basculé en mode évaluation. Le contexte `with torch.no_grad()` désactive temporairement le moteur de calcul automatique des dérivées (*autograd*), réduisant la consommation mémoire et accélérant l'inférence sur l'ensemble de test (`X_teste_t`). L'opération `.argmax(dim=1)` extrait la classe de plus haute probabilité pour chaque échantillon, permettant de calculer la précision sur le test.

3. **Visualisation avec la Bibliothèque `morph`:** La fonction `mm.showTrainCurves` de la bibliothèque didactique `morph` consolide l'historique de perte d'entraînement et la précision sur le test dans un unique panneau graphique, permettant de diagnostiquer la convergence du modèle et de surveiller la stabilité de l'apprentissage au fil des époques.

```
# 4. Boucle d'entraînement (SGD par mini-lots)
n = X_treino_t.size(0)                    # Nombre d'échantillons
tam_lote = 32                             # Taille du mini-lot
epocas = 50                               # Nombre total d'époques
historico_perda, historico_acc = [], []   # Historique des métriques

for epoca in range(epocas):                  # Répète par époque
    modelo_cnn.train()                       # Mode entraînement
    perm = torch.randperm(n)                 # Mélange les échantillons
    perda_epoca = 0.0                        # Accumule les pertes
    for i in range(0, n, tam_lote):          # Parcourt les mini-lots
        idx = perm[i:i + tam_lote]           # Indices du lot
        otimizador.zero_grad()               # Remet les gradients à zéro
        saida = modelo_cnn(X_treino_t[idx])  # Propagation avant
        perda = criterio(saida, y_treino_t[idx]) # Calcule la perte
        perda.backward()                         # Rétropropagation
        otimizador.step()                        # Met à jour les poids
        perda_epoca += perda.item() * len(idx)   # Somme des pertes

    # Évaluation du modèle sur l'ensemble de test à la fin de chaque époque
    modelo_cnn.eval()                            # Mode évaluation
    with torch.no_grad():                        # Sans gradients
        pred_teste = modelo_cnn(X_teste_t).argmax(dim=1)  # Prédictions
        acc_teste = (pred_teste == y_teste_t).float().mean().item()  # Précision
    historico_perda.append(perda_epoca / n)      # Enregistre la perte
    historico_acc.append(acc_teste)              # Enregistre la précision

acc_final_cnn = historico_acc[-1]                # Dernière précision
print(f"Précision finale du CNN sur l'ensemble de test : {acc_final_cnn:.4f}")  # Affiche le
résultat

final = mm.showTrainCurves(                      # Trace les courbes
    historico_perda, historico_acc,
    titulo="Evolução do Treinamento da CNN - Base de Dígitos",
    subtitulo=f"Acurácia final no teste: {acc_final_cnn:.4f}",
)
```

```
Précision finale du CNN sur l'ensemble de test : 0.9759
```

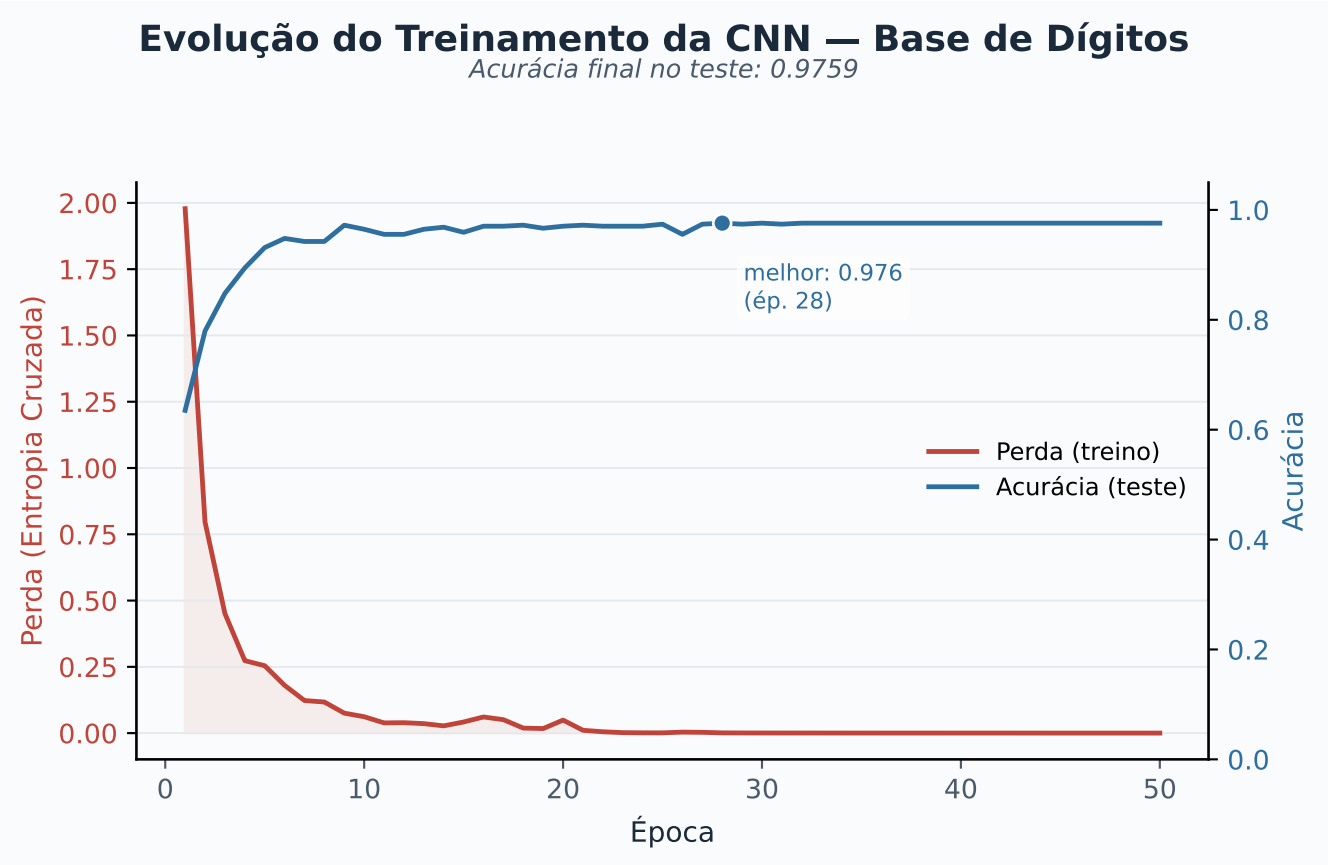


Figure 10.15: Courbes d'entraînement et d'évaluation du CNN sur la base de chiffres : évolution de la perte d'entropie croisée sur l'ensemble d'entraînement et de la précision sur l'ensemble de test au fil de 50 époques.

Bloc 5 : Visualisation du flux d'activations

L'inspection du réseau entraîné permet d'observer la transformation progressive du *tensor* d'entrée à travers les couches de l'architecture `CNNDigitos`. La Figure 10.16 illustre les dimensions et les activations intermédiaires obtenues lors du traitement d'un exemple réel du chiffre 3.

1. **Sélection et préparation de l'échantillon :** La graine stochastique est fixée avec `torch.manual_seed(7)` pour assurer la reproductibilité des résultats. La première occurrence du chiffre 3 dans l'ensemble de données `load_digits` est isolée, normalisée dans l'intervalle $[0.0, 1.0]$ et reconfigurée comme un *tensor* `x` de dimension $(1, 1, 8, 8)$.

2. **Inspection intermédiaire avec `mm.showNet` :** La fonction `mm.showNet` de la bibliothèque `morph` exécute la propagation avant (*forward pass*) du *tensor* `x` dans l'instance `modelo_cnn` préalablement entraînée. En utilisant des *hooks* de *forward*, la fonction intercepte l'état numérique des activations dans les couches convolutionnelles (`nn.Conv2d`), de regroupement (`nn.MaxPool2d`) et entièrement connectées (`nn.Linear`), et les retourne dans le dictionnaire `acts`. Les fonctions d'activation non linéaires (`nn.ReLU`) ne sont pas enregistrées comme étapes indépendantes, car leur application s'effectue directement sur le *tensor* de sortie de la couche correspondante.

3. **Vérification des résultats :** L'instruction `list(acts.keys())` affiche la séquence d'identifiants des couches surveillées, permettant de confirmer la réduction dimensionnelle progressive et la génération du *logit* de valeur maximale à l'indice correspondant à la classe 3, comme démontré dans la Figure 10.16..

```
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)

# Réutilisation de l'instance du modèle préalablement entraînée
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("Couches capturées :", list(acts.keys()))
```

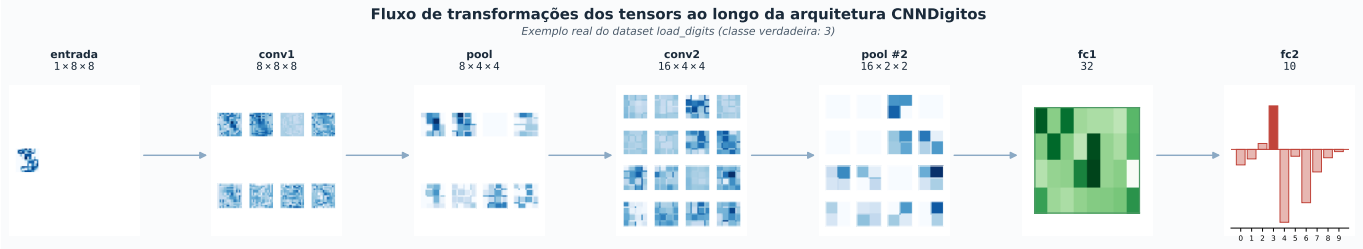


Figure 10.16: Fluxo de activations de la CNN entraînée lors du traitement d'un exemple réel du chiffre 3, du *dataset load_digits* : dimensions des tenseurs couche par couche, de l'entrée au *logit* de sortie.

```
Couches capturées : ['conv1', 'pool', 'conv2', 'pool #2', 'fc1', 'fc2']
```

Inspection du graphe computationnel avec `torchviz`

Alors que `mm.showNet` privilégie la clarté didactique — en affichant une colonne par couche avec les paramètres entraînables —, la bibliothèque `torchviz` projette le **graphe d'autograd** exactement comme PyTorch le construit en interne pour le calcul des gradients. La Figure 10.17 illustre cette perspective en représentant l'architecture `CNNDigitos`.

1. **Propagation avant tracée :** Avec le modèle entraîné en mode `eval()`, le *forward pass* sur le *tensor* `x` du chiffre 3 suffit pour que le moteur d'*autograd* enregistre toutes les opérations exécutées, y compris celles sans paramètres entraînables, comme la fonction d'activation `ReLU` et la reconfiguration dimensionnelle `view`.

2. **Génération du graphe (`make_dot`):** La fonction `make_dot(saida, params=...)` construit le graphe à partir du *tensor* de sortie, en parcourant rétroactivement l'historique des opérations jusqu'aux nœuds feuilles (les paramètres entraînables du modèle). Chaque nœud du diagramme représente une opération du *backward pass* (comme `ReluBackward` ou `AddmmBackward`), et non seulement un bloc conceptuel du `nn.Module`.

3. **Exportation et rendu (`.render`):** La méthode `.render(..., format="png", cleanup=True)` invoque l'exécutable `dot` de **Graphviz** pour compiler l'image au format PNG, en supprimant automatiquement les fichiers intermédiaires de code source.

La Figure 10.17 met en évidence comment ce graphe computationnel, même pour une architecture compacte, présente une densité plus élevée que le panneau du `mm.showNet`, car il détaille chaque opération atomique responsable du flux des gradients.

```
# 1. Passe avant avec suivi de gradient activé
modelo_cnn.eval()
saida = modelo_cnn(x)  # Réutilisation du tenseur x (chiffre 3)

# 2. Graphe de base : flux d'opérations jusqu'à la sortie
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. Affichage direct dans l'environnement Quarto/Jupyter
# Le .render() retourne le chemin du fichier PNG généré ; nous devons l'ouvrir comme image
imagem_simples = np.array(Image.open(caminho_simples).convert("RGB"))
mm.show(imagem_simples, figsize=(5,10))
```

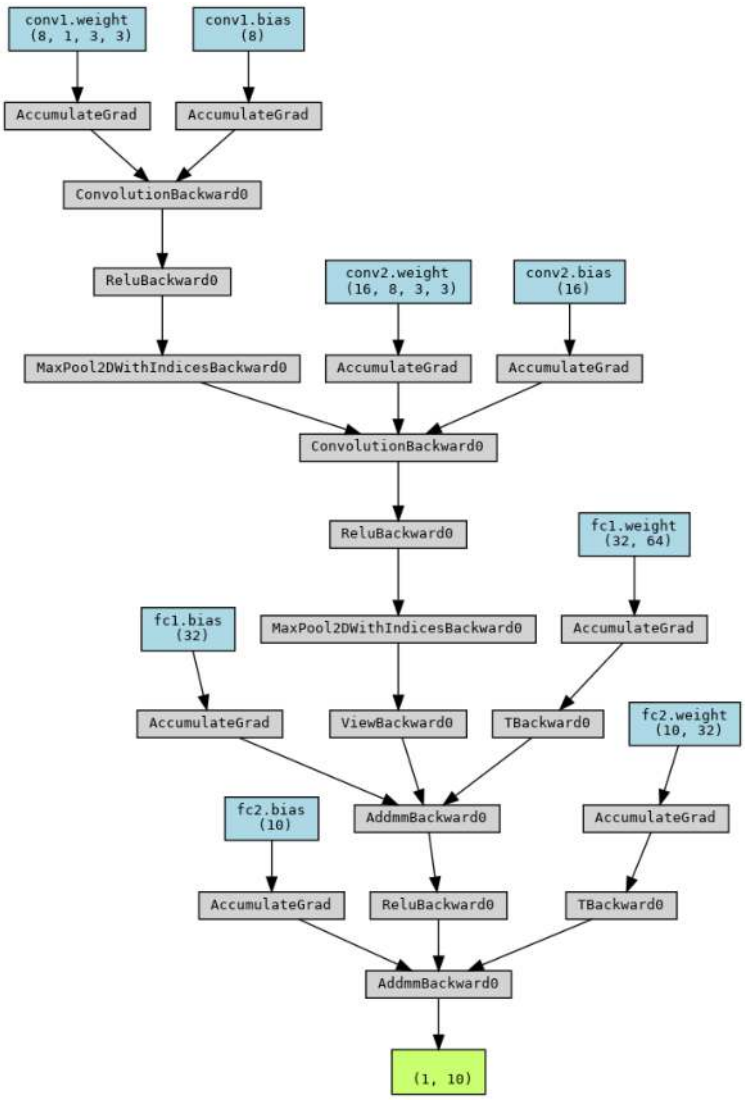


Figure 10.17: Graphe computationnel de la *CNNDigitos* généré via *torchviz*, montrant les opérations de *forward* et les nœuds de gradient (*backward*) associés à chaque paramètre entraînable.

Vue détaillée du graphe computationnel avec `torchviz`

Outre la représentation simplifiée, la bibliothèque `torchviz` permet d'étendre le graphe d'*autograd* afin d'inspecter les détails internes de l'exécution du réseau `CNNDigitos`. La Figure 10.18 présente cette structure étendue pour le même *tensor* d'entrée `x`.

1. **Traçage avec attributs d'opération (`show_attrs=True`):**
   L'inclusion des attributs affiche les configurations hyperparamétriques associées à chaque nœud computationnel durant la propagation avant (*forward pass*), telles que les dimensions de *kernel* (`kernel_size`), les pas (*stride*) et les remplissages (*padding*) dans les convolutions et les sous-échantillonnages.

2. **Détection des *tensors* sauvegardés en mémoire (`show_saved=True`):**

Ce paramètre force l'affichage explicite des *tensors* intermédiaires que PyTorch conserve en mémoire pendant le *forward pass*. Ces données sont préservées car elles seront strictement nécessaires au calcul des dérivées partielles durant l'étape de rétropropagation (*backward pass*).

3. **Génération et compilation des graphes:**
   Alors que `grafo_simples` produit une vue directe du flux de gradients, `grafo_detalhado` compile le graphe étendu dans le fichier `cnn_digitos_grafo_detalhado.png` via l'exécutable `dot` de **Graphviz**.

Comme observé dans la Figure 10.18, cette visualisation minutieuse est utile pour déboguer la consommation de mémoire vidéo (*VRAM*) et vérifier comment le moteur de PyTorch alloue en interne chaque nœud de la règle de la chaîne.

```
# même code précédent (make_dot + render)...

# 2. Graphe détaillé : affichage des dimensions et des tenseurs sauvegardés pour la
rétropropagation
grafo_detalhado = make_dot(
    saida,
    params=dict(modelo_cnn.named_parameters()),
    show_attrs=True,   # Affiche les attributs des opérations (ex. : kernel_size, stride)
    show_saved=True,   # Affiche les tenseurs sauvegardés en mémoire pour la rétropropagation
)

caminho_detalhado = grafo_detalhado.render("cnn_digitos_grafo_detalhado",
                                           format="png", cleanup=True)

# 3. Affichage direct dans l'environnement Quarto/Jupyter
imagem_detalhado = np.array(Image.open(caminho_detalhado).convert("RGB"))
mm.show(imagem_detalhado, figsize=(8, 16))
```

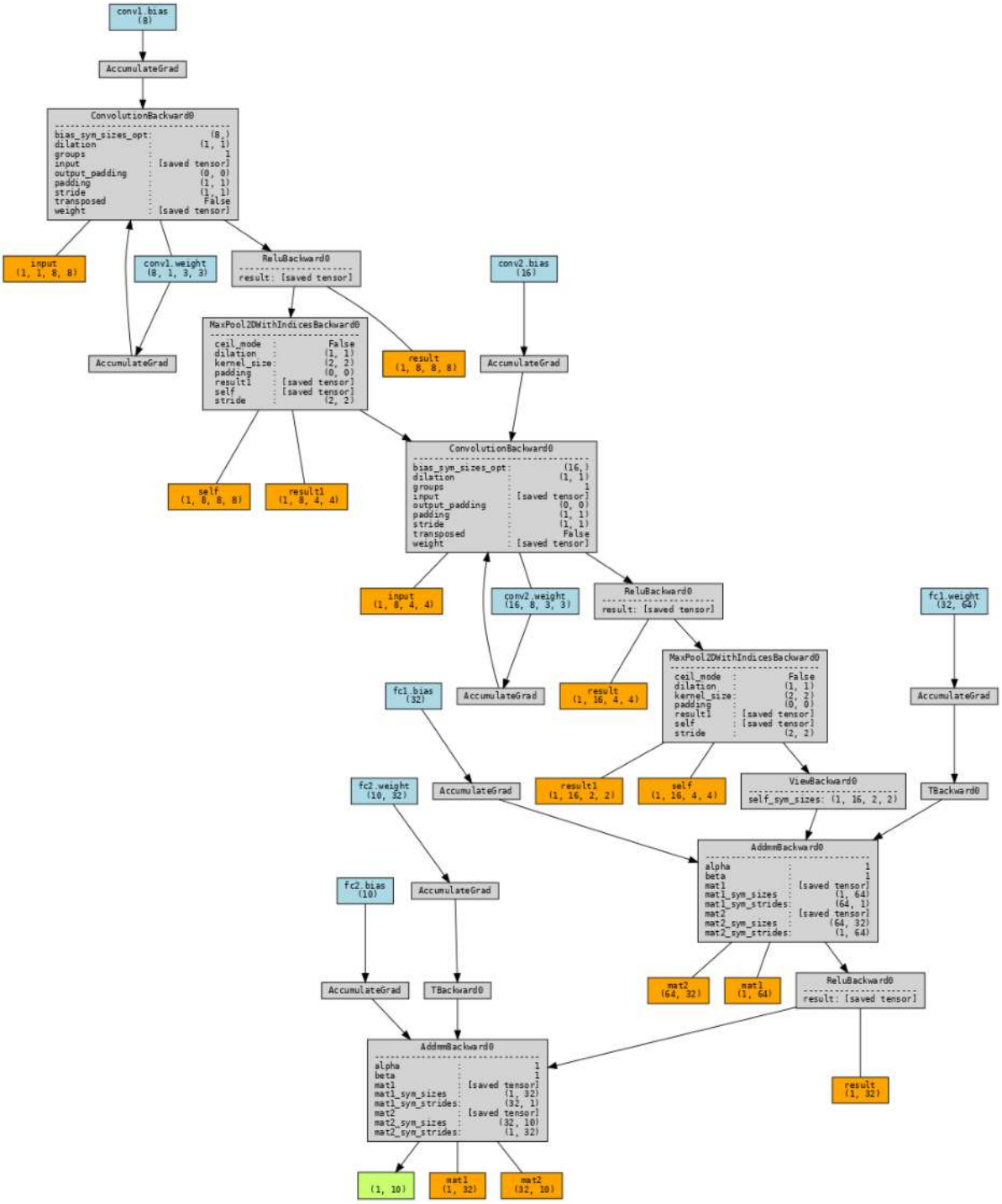


Figure 10.18: Graphe détaillé de la classe CNNDigitos généré via *torchviz*.

Comparaison avec le chapitre 7

La Figure 10.19 rassemble les résultats obtenus sur le même ensemble de données (`load_digits`), établissant un parallèle direct entre les approches classiques explorées précédemment et le CNN développé dans ce chapitre.

1. **Performance des pixels bruts vs. descripteurs manuels :** Dans les expériences du chapitre 7, le classificateur k-NN ($k = 3$) a atteint une précision de $98,4\%$ lorsqu'il était alimenté directement avec les pixels bruts des images. En revanche, l'extraction préalable de caractéristiques via *Histogram of Oriented Gradients* (*HOG*) a abouti à une performance significativement inférieure ($75,8\%$). Cette baisse s'explique par le fait que le *HOG* a été conçu pour capturer les gradients de contours dans des

images de plus haute résolution ; dans des matrices de seulement 8×8 pixels, la résolution spatiale est insuffisante pour former des histogrammes d'orientation informatifs.

2. **Équivalence du CNN et apprentissage *end-to-end* :** Le réseau convolutif `CNNDigitos` atteint une performance compétitive de $97,6\%$, se rapprochant de la précision du k-NN avec pixels bruts sur une base de données petite et pré-alignée. Le grand avantage conceptuel réside dans l'apprentissage de représentations : au lieu de dépendre de descripteurs conçus manuellement (*handcrafted features*) ou de conserver l'intégralité de l'ensemble de données en mémoire pour la recherche de voisins au moment de l'inférence, le CNN optimise automatiquement ses propres filtres convolutifs pendant l'entraînement, générant un modèle compact capable d'effectuer l'extraction de caractéristiques et la classification de manière intégrée (*end-to-end*).

```
import matplotlib.pyplot as plt

# Valeurs obtenues au Chapitre 7 (k-NN, k=3), reproduites pour comparaison directe
ACC_KNN_PIXELS_CAP7 = 0.9844
ACC_KNN_HOG_CAP7 = 0.7578

metodos = ["k-NN\n(pixels brutos)", "k-NN\n(HOG)", "CNN\n(este capítulo)"]
acuracias = [ACC_KNN_PIXELS_CAP7, ACC_KNN_HOG_CAP7, acc_final_cnn]

plt.figure(figsize=(5, 4))
cores = ["#6366f1", "#f97316", "#16a34a"]
plt.bar(metodos, acuracias, color=cores)
plt.ylim(0, max(acuracias) + 0.08)
plt.ylabel("Acurácia (conjunto de teste)")
plt.title("Cap. 7 vs. Cap. 9 - Base de Dígitos")

for i, v in enumerate(acuracias):
    plt.text(i, v + 0.02, f"{v:.3f}", ha="center")

plt.tight_layout()
plt.show()
```

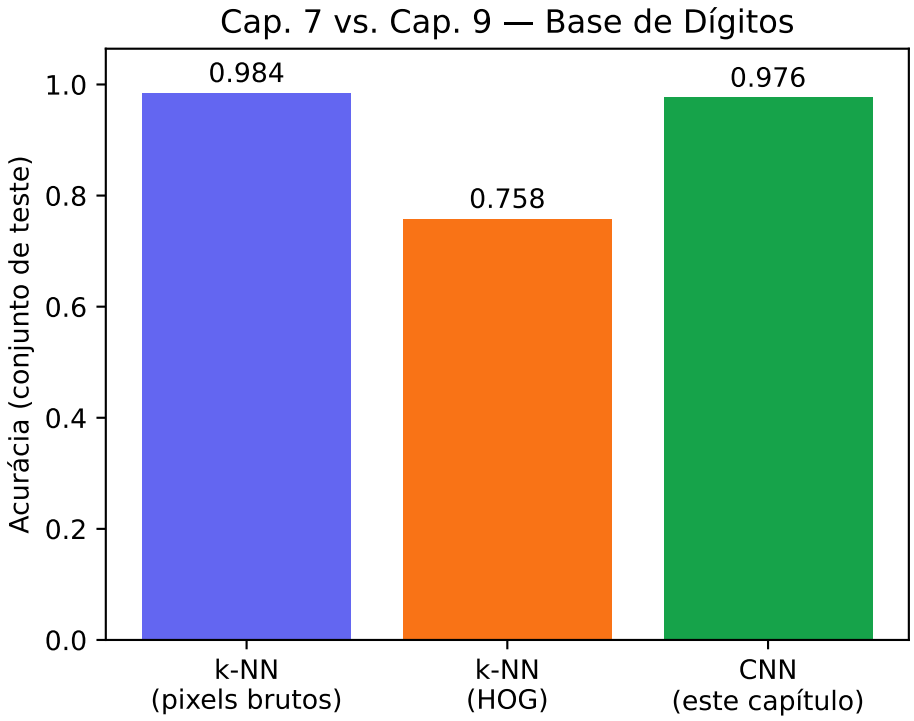


Figure 10.19: Comparaison de précision entre les classificateurs classiques du Chapitre 7 (pixels bruts et HOG avec k-NN) et le CNN entraîné dans ce chapitre, sur la même base de chiffres.

ℹ 🧠 Pourquoi cela fonctionne-t-il ? — Et pourquoi le CNN ne « gagne » pas toujours

Le résultat observé ici **répète le schéma déjà vu au Chapitre 7** : le CNN, bien qu'il apprenne automatiquement ses caractéristiques, ne surpasse pas nécessairement le k-NN avec des *pixels* bruts sur cette base spécifique. L'explication est la même : `load_digits` est une base petite (moins de 1.800 exemples), avec des images déjà centrées, normalisées et de très basse résolution (8×8) — des conditions où la comparaison directe des intensités est déjà hautement informative, et où il y a peu de données pour que le réseau apprenne des filtres véritablement supérieurs aux descripteurs simples.
Le véritable avantage des CNN apparaît dans des scénarios que les descripteurs artisanaux et les classificateurs simples ne peuvent pas aborder : des images plus grandes et plus réalistes, avec des milliers de catégories, une variation substantielle de pose, d'éclairage et de fond, et des ensembles d'entraînement massifs — exactement le régime dans lequel les modèles présentés dans la section *« Applications à Grande Échelle »*, plus loin, ont été entraînés. La leçon pédagogique qui traverse les Chapitres 7, 8 et 9 de ce livre est cohérente : **la sophistication d'une méthode doit être proportionnelle à la complexité du problème** — utiliser un CNN pour un problème qu'un k-NN résout tout aussi bien est un gaspillage de ressources computationnelles, non une vertu.
Cette même proportionnalité vaut pour les outils d'inspection utilisés tout au long du chapitre. Le `mm.showNet` a été construit à des fins **didactiques** et fonctionne bien sur des réseaux peu profonds comme `CNNDigitos`, mais ne passe pas à l'échelle pour des architectures profondes : chaque couche suivie devient une colonne dans la figure, et les couches convolutionnelles avec des centaines de canaux génèrent des mosaïques trop grandes pour une interprétation visuelle ; de plus, les *hooks* stockent toutes les activations en mémoire, et la *mise en page* suppose un flux séquentiel, ne représentant pas fidèlement les connexions résiduelles ou les ramifications (comme dans les *ResNets* ou les modules *Inception*). Ainsi, `showNet` doit être compris comme une lentille pédagogique pour les petits réseaux — analogue au rôle de `mm.showBoundBox` dans le débogage visuel des détections — et non comme un substitut aux outils orientés production, comme *TensorBoard* ou *torchviz*.

10.5.1.2 Transfert d'apprentissage

Entraîner un CNN de zéro nécessite généralement une grande quantité de données étiquetées et des ressources computationnelles significatives, car le processus d'entraînement doit ajuster tous les paramètres du réseau. Dans de nombreuses applications, cependant, seul un ensemble réduit de données est disponible pour la tâche d'intérêt. Dans cette situation, le **transfert d'apprentissage** (*transfer learning*) réutilise les représentations apprises par un modèle préalablement entraîné sur une tâche source avec un grand volume de données, réduisant ainsi le coût d'entraînement et le besoin de nouvelles échantillons.

En vision par ordinateur, cette stratégie exploite l'organisation hiérarchique des CNN. Les couches initiales apprennent des caractéristiques visuelles de bas niveau, telles que les contours, les textures, les gradients d'intensité et les motifs de couleurs, qui restent utiles dans différents domaines. Les couches plus profondes combinent ces informations pour former des représentations progressivement plus abstraites et spécialisées, liées aux classes présentes dans la base d'entraînement.

Cette section examine dans quelles conditions le transfert d'apprentissage produit de bons résultats. La première expérience montre qu'un extracteur de petite taille et entraîné sur un domaine restreint peut conduire à un **transfert négatif** (*negative transfer*). La deuxième démontre pourquoi des modèles profonds pré-entraînés sur de grandes bases d'images atteignent des performances élevées sur de nouvelles tâches. Enfin, la troisième applique cette stratégie à un problème de diagnostic phytosanitaire, illustrant un scénario proche des applications réelles.

Expérience 1 — Limites d'un extracteur petit et spécialisé

La première expérience montre que le transfert d'apprentissage n'améliore pas toujours les performances d'un modèle. Pour ce faire, l'ensemble de chiffres manuscrits (`load_digits`) est divisé en deux domaines disjoints :

- **Domaine A (source) :** chiffres 0 à 4, utilisés pour entraîner une petite CNN ;
- **Domaine B (cible) :** chiffres 5 à 9, réindexés de 0 à 4, formant une nouvelle tâche avec seulement 20 échantillons d'entraînement.

L'objectif consiste à évaluer l'effet de la réutilisation de l'extracteur de caractéristiques appris dans le Domaine A sans permettre son adaptation au Domaine B.

Bloco 1 : Division des domaines, rareté et affichage des échantillons

Ce bloc prépare l'ensemble de données pour l'expérience. Contrairement au projet précédent, qui utilisait tous les chiffres dans un problème de classification unique, la base est divisée en deux tâches indépendantes : une tâche source (Domaine A) et une tâche cible (Domaine B).

La Figure 10.20 présente des exemples des deux domaines après la séparation des classes, la conversion en *tensors* **PyTorch** et le prétraitement.

1. **Séparation des classes :** Les masques booléens `mask_A` et `mask_B` séparent les exemples de chaque domaine. Ensuite, le code réindexe les étiquettes du Domaine B (`y[mask_B] - 5`) dans l'intervalle $[0, 4]$, permettant ainsi aux deux modèles d'utiliser cinq classes de sortie.

2. **Rareté des données :** Le générateur `np.random.default_rng(0)` sélectionne uniquement 20 échantillons pour l'entraînement du Domaine B, environ quatre par classe, simulant un scénario où l'entraînement à partir de zéro tend à souffrir d'*overfitting*.

3. **Conversion en *tensors* :** La fonction `para_tensor` convertit les images au format $(N, 1, 8, 8)$ et les étiquettes en `torch.long`, compatibles avec les couches `nn.Conv2d` et la fonction de perte.

4. **Visualisation des échantillons :** Le code utilise `.squeeze().numpy()` pour convertir les *tensors* en matrices **NumPy**. La Figure 10.20 présente des exemples des deux domaines et met en évidence la réindexation appliquée aux étiquettes du Domaine B.

```
# 1. Division du jeu de données en deux domaines disjoints
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  # Réindexation des étiquettes dans l'intervalle [0, 4]

# Division en entraînement et test pour les deux domaines
XA_tr, XA_te, yA_tr, yA_te = train_test_split(
    XA, yA, test_size=0.25, random_state=42, stratify=yA
)
XB_tr, XB_te, yB_tr, yB_te = train_test_split(
    XB, yB, test_size=0.25, random_state=42, stratify=yB
)

# Simulation de pénurie extrême dans le domaine cible : seulement 20 échantillons
d'entraînement
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]

# Fonction auxiliaire pour la conversion en tenseurs 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)

# Affichage d'échantillons des deux domaines
n_amostras = 5
imgs_A = [img.squeeze().numpy() for img in XA_tr_t[:n_amostras]]
titles_A = [f"A: {label.item()}" for label in yA_tr_t[:n_amostras]]

imgs_B = [img.squeeze().numpy() for img in XB_tr_t[:n_amostras]]
titles_B = [f"B: {label.item()} (orig: {label.item()+5})" for label in yB_tr_t[:n_amostras]]

mm.show(
    imgs_A + imgs_B,
    titles=titles_A + titles_B,
    cols=n_amostras,
    figsize=(12, 4.5)
)
```

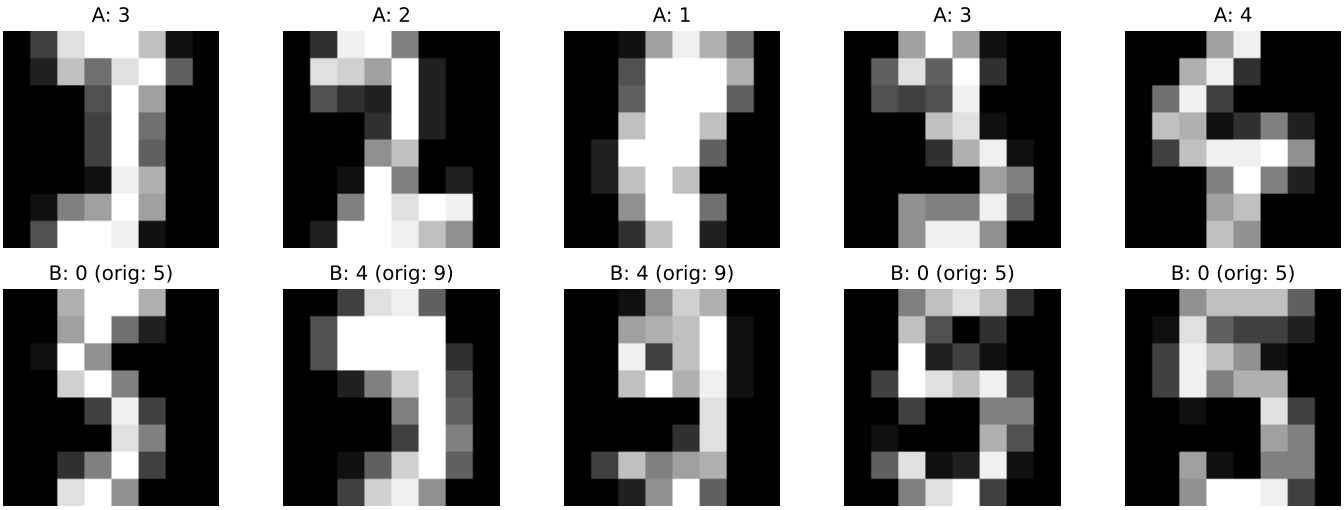


Figure 10.20: Échantillons des ensembles d'entraînement après prétraitement et ajustement dimensionnel aux tenseurs PyTorch : Domaine A (chiffres 0 à 4, tâche source) et Domaine B (chiffres 5 à 9 réindexés en 0 à 4, tâche cible).

Bloc 2 : Architecture Modulaire et Routines Génériques

Pour faciliter le transfert d'apprentissage, l'architecture convolutive et la boucle d'entraînement ont été refactorisées par rapport à la classe `CNNDigitos` du projet précédent.

1. **Modularisation de l'architecture (Différence par rapport à `CNNDigitos`) :**
   - Dans le projet précédent, la classe `CNNDigitos` déclarait toutes les couches (`conv1`, `conv2`, `pool`, `fc1`, `fc2`) comme membres directs d'une seule classe monolithique.
   - Ici, l'architecture est séparée en deux composants : la classe `ExtratorConv` encapsule le bloc spatial convolutif (2 convolutions 3×3, 2 *Max-Poolings* 2×2 et l'aplatissement en 64 éléments), tandis que la classe `CNNCompleta` instancie cet extracteur dans `self.extrator` et ajoute la « tête » de classification (`fc1` et `fc2`).
   - Cette séparation est ce qui permet de copier l'état interne de l'extracteur (`state_dict()`) d'un modèle à un autre de manière isolée.
2. **Ajustement du nombre de classes de sortie :** Alors que `CNNDigitos` dans le projet précédent possédait 10 *logits* dans la couche de sortie (`self.fc2 = nn.Linear(32, 10)`), la classe `CNNCompleta` reçoit `n_classes=5` dans le constructeur pour s'adapter à la division des domaines A et B.
3. **Flexibilisation de la boucle d'entraînement (`treinar`) :**
   - Dans le projet précédent, la boucle d'entraînement itérait directement sur les attributs globaux du modèle (`modelo_cnn.parameters()`) et calculait des métriques spécifiques en ligne.
   - La fonction `treinar` abstrait ce processus et introduit le paramètre optionnel `parametros`. S'il est fourni, l'optimiseur *Adam* met à jour **uniquement** les paramètres de cette liste, ignorant les couches dont les gradients ont été désactivés. Cette flexibilité est cruciale pour exécuter l'entraînement avec un gel partiel du réseau.
4. **Isolation de l'évaluation (`calcular_acuracia`) :** Comme cela a été fait lors de la phase de test du projet précédent, la fonction place le modèle en mode `eval()` et utilise le contexte `torch.no_grad()` pour désactiver l'*autograd*, calculant la précision via `.argmax(dim=1)`.

```
# Définition du bloc convolutif réutilisable (même extraction que le projet précédent)
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)

# Architecture modulaire combinant l'extracteur et la tête de classification
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)

# Routine générique d'entraînement avec optimisation sélective des paramètres
def treinar(modelo, X_t, y_t, epocas, lr, tam_lote=16, parametros=None):
    # Paramètres entraînables
    params = parametros if parametros is not None else modelo.parameters()
    otim = optim.Adam(params, lr=lr)                # Optimiseur Adam
    crit = nn.CrossEntropyLoss()                    # Fonction de perte
    n_amostras = X_t.size(0)                        # Nombre d'échantillons
    for _ in range(epocas):                         # Répète par époque
        perm = torch.randperm(n_amostras)           # Mélange les échantillons
        for i in range(0, n_amostras, tam_lote):    # Parcourt les mini-lots
            idx = perm[i:i + tam_lote]              # Indices du lot
            otim.zero_grad()                        # Met à zéro les gradients
            perda = crit(modelo(X_t[idx]), y_t[idx])  # Calcule la perte
            perda.backward()                        # Rétropropagation
            otim.step()                             # Met à jour les poids

# Routine d'évaluation
def calcular_acuracia(modelo, X_t, y_t):
    modelo.eval()                                   # Mode évaluation
    with torch.no_grad():                           # Sans gradients
        pred = modelo(X_t).argmax(dim=1)            # Classes prédites
    return (pred == y_t).float().mean().item()      # Retourne la précision
```

Bloco 3: Pré-entraînement, Transfert et Analyse Comparative

Ce bloc exécute la comparaison entre l'ajustement du modèle à partir de zéro et l'application du transfert avec gel statique de l'extracteur. Pour une transparence totale de l'expérience, les tailles des ensembles d'entraînement et de test des deux domaines sont imprimées dans le terminal.

1. **Quantification des échantillons par domaine :**

   - **Domaine A (source, chiffres 0 à 4):** Compte 675 échantillons d'entraînement (75%) et 226 de test (25%), fournissant des données abondantes pour que le `modelo_origem` apprenne l'extracteur convolutif jusqu'à atteindre 100% de précision.
   - **Domaine B (cible, chiffres 5 à 9):** Dispose de 224 échantillons de test au total, mais son ensemble d'entraînement est intentionnellement réduit de 672 à seulement 20 **échantillons**

(`XB_tr_poucos`), créant un scénario sévère de pénurie de données.

2. **Étape 1 : Pré-entraînement sur le Domaine A (Source) :** Le `modelo_origem` est entraîné à partir de zéro sur les 675 échantillons des chiffres 0 à 4. Pendant 40 époques, l'extracteur convolutif ajuste ses filtres pour identifier les traits caractéristiques de ces cinq premiers chiffres, atteignant 100% de précision sur l'ensemble de test (226 échantillons).

3. **Étape 2 : Transfert de poids et gel :**
   - On crée le `modelo_transferencia` pour résoudre la tâche du Domaine B (chiffres 5 à 9).
   - Les poids appris dans le Domaine A sont copiés via :
     - `load_state_dict(modelo_origem.extrator.state_dict())`
   - **Gel :** La boucle `for p in modelo_transferencia.extrator.parameters(): p.requires_grad = False` désactive le calcul des gradients dans les couches convolutives.
   - **Entraînement sélectif :** L'appel `treinar(...)` passe strictement les paramètres des couches denses (`params_cabeca`), ajustant la tête de classification avec seulement les 20 échantillons d'entraînement.

4. **Étape 3 : Entraînement à partir de zéro sur le Domaine B (Contrôle expérimental) :** Le `modelo_do_zero` possède la même architecture, mais est entraîné à partir de zéro sur les mêmes 20 échantillons du Domaine B, sans aucune réutilisation de poids, pendant les mêmes 40 époques.

5. **Analyse des résultats (Figure 10.21) :**
   - **Avec transfert gelé ($72,77\%$):** En réutilisant l'extracteur entraîné sur le Domaine A et en gelant ses paramètres, le réseau atteint $72,77\%$ de précision sur le test (224 échantillons) en ajustant uniquement les couches denses.
   - **Entraîné à partir de zéro ($76,79\%$):** L'entraînement à partir de zéro surpasse le transfert gelé sur l'ensemble de test du Domaine B.
   - **Cause de la différence :** En raison d'un modèle minuscule (seulement 16 filtres convolutifs dans des matrices de 8×8), l'extracteur entraîné sur le Domaine A est devenu **hyperspécialisé** dans les formes géométriques des chiffres 0 à 4. En gelant rigidement ces quelques filtres, le modèle cible s'est trouvé limité à des détecteurs inadéquats pour 5 à 9. Le réseau entraîné à partir de zéro, même avec seulement 20 échantillons, a réussi à adapter ses 16 filtres directement aux traits du Domaine 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
```

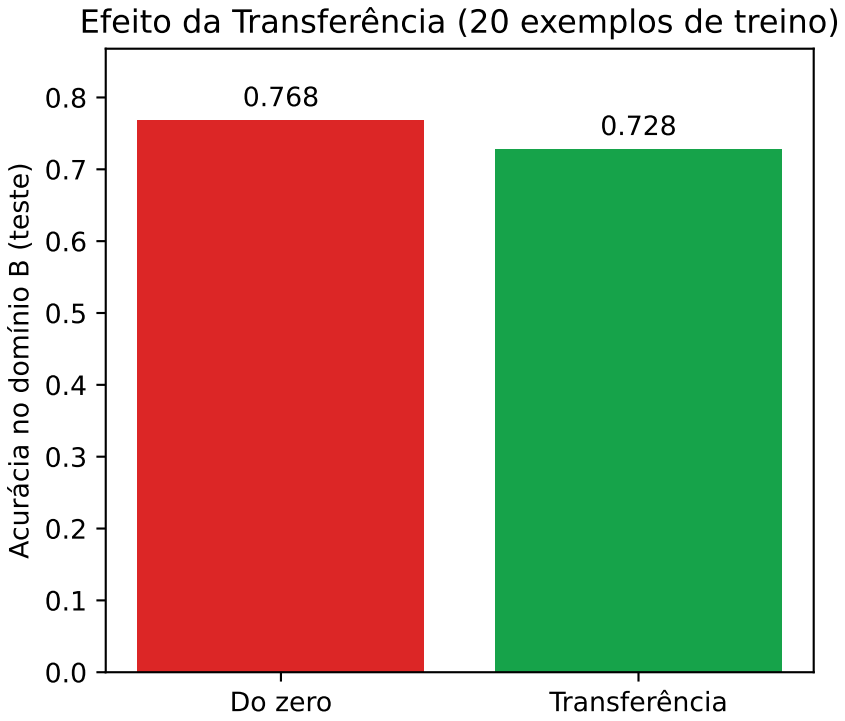


Figure 10.21: Comparação de acurácia no conjunto de teste do Domínio B (dígitos 5 a 9) sob restrição de dados (20 exemplos de treino): demonstração do impacto do congelamento rígido e da transferência negativa em redes de baixa capacidade.

Bloco 4 : Visualisation du flux d'activations avec `mm.showNet`

Pour confirmer que l'extraction de caractéristiques réutilisée préserve les transformations dimensionnelles étudiées dans le projet précédent, on utilise à nouveau la fonction `mm.showNet` de la bibliothèque `morph`. La Figure 10.22 affiche le flux d'activations du `modelo_transferencia` lors du traitement d'un échantillon du Domaine B (chiffre 7, réindexé pour la classe 2).

1. **Préservation du flux convolutionnel :** Comme l'architecture `ExtratorConv` reproduit les mêmes couches de convolution et de *pooling* que la `CNNDigitos` du projet précédent, les dimensions des *tensors* intermédiaires restent en $(1, 8, 4, 4)$ dans le premier bloc.
2. **Inspection de la tête adaptée :** La différence par rapport au projet précédent apparaît dans la couche de sortie (`fc2`) : alors que le modèle du projet précédent projetait le vecteur intermédiaire en 10 *logits* (classes de 0 à 9), le modèle de transfert projette le vecteur en 5 *logits* (classes de 0 à 4), capturant les probabilités relatives du Domaine B.

```
# Sélectionne le premier échantillon de test du Domaine B
x_amostra_B = XB_te_t[0:1]  # Tenseur de dimension (1, 1, 8, 8)
classe_verdadeira = yB_te_t[0].item()
classe_original = classe_verdadeira + 5

# Inspection du flux d'activations dans le modèle de transfert
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("Couches capturées dans le modèle de transfert :\n", list(acts_transfer.keys()))
```

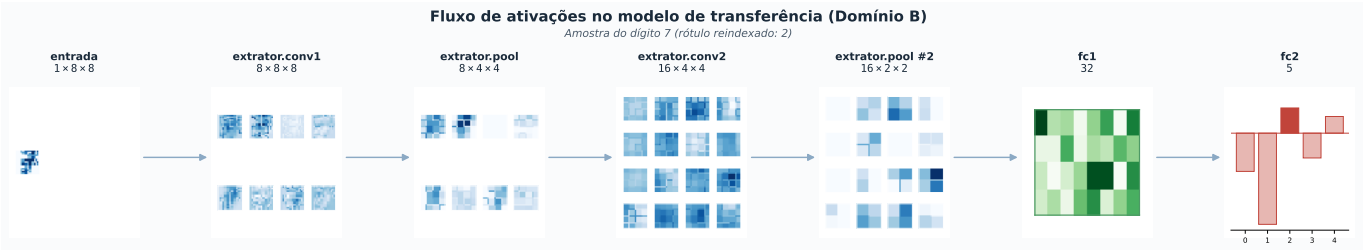


Figure 10.22: Flux d'activations et transformations dimensionnelles des tenseurs dans le modèle de transfert d'apprentissage lors du traitement d'un échantillon de test du Domaine B (chiffre 7, réindexé en classe 2).

```
Couches capturées dans le modèle de transfert :
 ['extrator.conv1', 'extrator.pool', 'extrator.conv2', 'extrator.pool #2', 'fc1', 'fc2']
```

Analyse de l'expérience 1

Le modèle entraîné de zéro atteint une précision supérieure à celle du modèle avec transfert d'apprentissage et extracteur figé. Ce résultat caractérise un cas de **transfert négatif** (*negative transfer*) et découle de trois facteurs :

1. **Faible capacité :** L'extracteur ne possède que 16 filtres 3×3, insuffisants pour apprendre des représentations généralisables.
2. **Spécialisation au domaine :** L'entraînement avec les chiffres 0 à 4 produit des filtres peu discriminants pour les chiffres 5 à 9.
3. **Absence d'adaptation :** Le gel empêche l'extracteur d'ajuster ses filtres à la nouvelle tâche.

Provocation pédagogique

Cette expérience utilise un petit extracteur entraîné sur un domaine restreint. Le résultat serait-il différent si l'extracteur avait appris ses représentations sur une base de millions d'images et une grande diversité d'objets ?

Expérience 2 — Quand le Transfert Fonctionne Vraiment (*ResNet-18* Pré-entraînée)

La deuxième expérience reprend la même structure que la première — peu d'exemples d'entraînement, deux classes, comparaison entre stratégies —, mais remplace l'extracteur artisanal de 16 filtres par la ***ResNet-18***, une architecture de 18 couches pré-entraînée sur *ImageNet* ($1,4$ million d'images, 1.000 catégories), et le *dataset* synthétique de chiffres par des photographies réelles du **Oxford-IIIT Pet Dataset** (PARKHI, 2012).

La tâche : distinguer deux races canines — **Carlin** et **Boxer** — à partir de seulement 15 photographies d'entraînement par classe.

Pourquoi ce scénario ?

Le défi ici ne réside pas dans la similarité visuelle entre les races — le Carlin et le Boxer ont des tailles et des proportions bien distinctes —, mais dans la rareté des données : seulement 30 photographies réelles au total, sans aucune image synthétique. C'est le type de problème à faible budget de données qui motive, en pratique, l'utilisation de réseaux pré-entraînés : il n'y a ni le temps ni les ressources pour photographier et étiqueter des milliers de chiens avant d'entraîner un classificateur à partir de zéro.

Bloc 1 : Chargement du Jeu de Données Réel et Échantillonnage Parcimonieux

1. **Source :** le *Oxford-IIIT Pet Dataset* (PARKHI, 2012) est chargé via `torchvision.datasets.OxfordIIITPet`, qui télécharge automatiquement les 7.349 photographies et leurs étiquettes de race lors de la première

exécution.
2. **Filtrage :** seules les deux races d'intérêt (`Pug`, `Boxer`) sont conservées.
3. **Parcimonie délibérée :** seules 15 photographies d'entraînement par classe (30 au total) sont tirées au sort — le reste constitue l'ensemble de test, utilisé exclusivement pour l'évaluation.

La Figure 10.23 affiche des échantillons d'entraînement de chaque race.

```
import random

RACAS_ALVO = ["Pug", "Boxer"]
N_TREINO_POR_CLASSE = 15
N_TESTE_POR_CLASSE = 20

# 1. Téléchargement du jeu de données complet (37 races) - licence 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. Filtrage des deux races d'intérêt, séparées par classe
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. Échantillonnage : peu d'images d'entraînement, plus d'images de test
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"Entraînement : {len(imgs_treino)} images | Test : {len(imgs_teste)} images")

amostras_pil = imgs_treino[:4] + imgs_treino[N_TREINO_POR_CLASSE:N_TREINO_POR_CLASSE + 4]
amostras_exibicao = [np.array(img.convert("RGB")) for img in amostras_pil]  # PIL -> ndarray
titulos_exibicao = [RACAS_ALVO[0]] * 4 + [RACAS_ALVO[1]] * 4
mm.show(amostras_exibicao, titles=titulos_exibicao, cols=4, figsize=(11, 6))
```

Figure 10.23

Bloc 2 : Trois stratégies sur la même architecture

Pour isoler l'effet du transfert d'apprentissage, les trois stratégies réutilisent **exactement la même architecture** (*ResNet-18*), en ne variant que l'origine des poids et les paramètres qui restent entraînables :

1. **`do_zero` :** poids aléatoires (`weights=None`) — équivalent à entraîner l'architecture de la *ResNet-18* entièrement de zéro, comme dans le Bloc 2 de l'Expérience 1.
2. **`congelado` :** poids pré-entraînés sur *ImageNet*, avec `requires_grad = False` sur toutes les couches convolutionnelles — seule la nouvelle couche finale est entraînée.
3. **`fine_tuning` :** poids pré-entraînés sur *ImageNet* comme point de départ, mais **sans** gel — tout le réseau s'adapte au nouveau domaine, avec un taux d'apprentissage faible pour ne pas détruire les connaissances préalables.

Dans tous les cas, la couche finale `fc` est remplacée par `nn.Linear(fc.in_features, 2)`, correspondant aux deux races cibles.

```
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")
```

Bloc 3 : Entraînement comparatif et analyse de la précision

En réutilisant les fonctions génériques `treinar` et `calcular_acuracia`, définies dans le Bloc 2 de l'Expérience 1, les trois modèles sont entraînés sur le même ensemble de 30 photographies et évalués sur l'ensemble de test (images jamais vues pendant l'entraînement) :

- Le modèle `do_zero` a tendance à **surajuster** rapidement aux 30 photographies d'entraînement, sans généraliser à l'ensemble de test — 30 exemples sont drastiquement insuffisants pour ajuster les 11 millions de paramètres de la *ResNet-18* à partir de zéro.
- Le modèle `congelado` devrait déjà atteindre une précision considérablement supérieure, car il réutilise, sans aucun ajustement, des caractéristiques visuelles génériques (bords, textures, contours) apprises sur *ImageNet* — seule la nouvelle couche linéaire doit être ajustée aux 30 photographies.
- Le modèle `fine_tuning` tend à égaler ou à dépasser l'extracteur figé, car il part des mêmes connaissances préalables, mais permet en outre un ajustement fin de tout le réseau aux particularités visuelles des races.

La Figure 10.24 résume les trois résultats.

Bloc 4 : Inspection qualitative des prédictions

Comme dans l'Expérience 1 et dans la section suivante sur le diagnostic foliaire, il est instructif d'observer individuellement certaines prédictions du meilleur modèle (typiquement `fine_tuning` ou `congelado`) sur des photographies réelles de test, en comparant l'étiquette prédite avec la race réelle.

```
# Nettoyage explicite des données téléchargées et libération de la mémoire
if FLAG_LIMPAR_DADOS:
    if os.path.exists('./dados_pets'):
        shutil.rmtree('./dados_pets')
        print('  Répertoire de données temporaires ./dados_pets supprimé avec succès.')
```

```
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")

plt.xticks(rotation=10)
plt.legend()
plt.tight_layout()
plt.show()
```

Figure 10.24

```
melhor_modelo = modelo_fine_tuning  # ou modelo_congelado, selon le résultat du Bloc 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))
```

Figure 10.25

```
    if torch.cuda.is_available():
        torch.cuda.empty_cache()
```

Expérience 3 — Diagnostic phytosanitaire par apprentissage par transfert

L'expérience précédente a montré qu'une *ResNet-18* pré-entraînée sur **ImageNet** peut s'adapter à une nouvelle tâche en utilisant peu d'échantillons. Désormais, la même stratégie est appliquée à un problème de diagnostic phytosanitaire. La *ResNet-18* doit classer des images de feuilles en trois catégories : `["folha_saudavel", "folha_doente", "sintoma_desconhecido"]`.

- **`folha_saudavel` :** feuille sans lésions visibles.
- **`folha_doente` :** feuille présentant des taches sombres simulant une maladie fongique.
- **`sintoma_desconhecido` :** feuille avec chlorose jaunâtre, représentant un motif différent de la maladie connue.

Pourquoi ce scénario ?

Le diagnostic phytosanitaire constitue une application importante de la vision par ordinateur (VC) dans l'agriculture de précision. Un modèle pré-entraîné sur **ImageNet** peut réutiliser des caractéristiques telles que les bords, les textures et les motifs de couleur pour apprendre cette nouvelle tâche avec peu d'images.

Bloc 1 : Génération du *dataset* synthétique

Ce bloc génère un ensemble synthétique avec **30** images par classe pour l'entraînement et **8** pour la validation, totalisant respectivement **90** et **24** images.

1. **Génération de la feuille :** La fonction `desenha_folha_base` crée le contour de la feuille, en variant la taille, l'orientation et la tonalité de vert.
2. **Simulation de la maladie :** La fonction `aplica_manchas_doenca` ajoute des taches sombres irrégulières simulant des lésions fongiques.
3. **Simulation d'un autre symptôme :** La fonction `aplica_sintoma_desconhecido` ajoute des régions jaunâtres représentant un motif distinct de la maladie connue.
4. **Visualisation des échantillons :** La Figure 10.26 présente des exemples des trois catégories de l'ensemble synthétique.

```
CLASSES_FOLHA = ["folha_saudavel", "folha_doente", "sintoma_desconhecido"]


def desenha_folha_base(tam_img, rng):
    '''Dessine le contour ovale d'une feuille verte avec nervure centrale,
    avec de petites variations de teinte, de taille et d'orientation entre les
échantillons.'''
    img = np.full((tam_img, tam_img, 3), 245, dtype=np.uint8)  # fond clair
    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)  # nervure centrale
    return img, (cx, cy, eixo_a, eixo_b, angulo)


def aplica_manchas_doenca(img, centro_folha, rng, n_manchas=(4, 8)):
    '''Simule des lésions foliaires : taches sombres aux bords irréguliers
    (motif typique des maladies fongiques).'''
    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):
    '''Simule un motif distinct (marbrures jaunâtres/chlorose), différent
    des taches sombres de la maladie connue.'''
    cx, cy, eixo_a, eixo_b, _ = centro_folha
    for _ in range(rng.randint(3, 5)):
        eixo_m = (rng.randint(1, 3), rng.randint(2, 3))
        px = cx + rng.randint(-int(eixo_a * 0.6), int(eixo_a * 0.6))
        py = cy + rng.randint(-int(eixo_b * 0.6), int(eixo_b * 0.6))
        cor_clorose = (rng.randint(200, 235), rng.randint(195, 225), rng.randint(50, 90))
        ang_m = rng.uniform(0, 180)
        cv2.ellipse(img, (px, py), eixo_m, ang_m, 0, 360, cor_clorose, -1, cv2.LINE_AA)
    return img


def aplica_ruido_sal_pimenta(img, prop_ruido=0.02, rng=None):
    '''Applique du bruit sel (points blancs) et poivre (points noirs) aléatoires.
    prop_bruit : fraction de pixels modifiés (ex : 0.02 = 2 % des pixels).'''
    if prop_ruido <= 0:
        return img

    img_ruido = img.copy()
    num_pixels = int(prop_ruido * img.shape[0] * img.shape[1])
    n_sal = num_pixels // 2
    n_pimenta = num_pixels - n_sal

    # Applique Sel (Blanc - [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]

    # Applique Poivre (Noir - [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)  # légère variation d'exposition
    img = np.clip(img.astype(np.int16) + ruido_exp, 0, 255).astype(np.uint8)

    # Application du bruit Sel et Poivre
    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"Entraînement : {len(imgs_treino)} images ({N_POR_CLASSE_TREINO} par classe) | "
      f"Validation : {len(imgs_val)} images ({N_POR_CLASSE_VAL} par classe)")

# Affichage de 2 échantillons de chaque classe (6 images au 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))
```

```
Entraînement : 90 images (30 par classe) | Validation : 24 images (8 par classe)
```

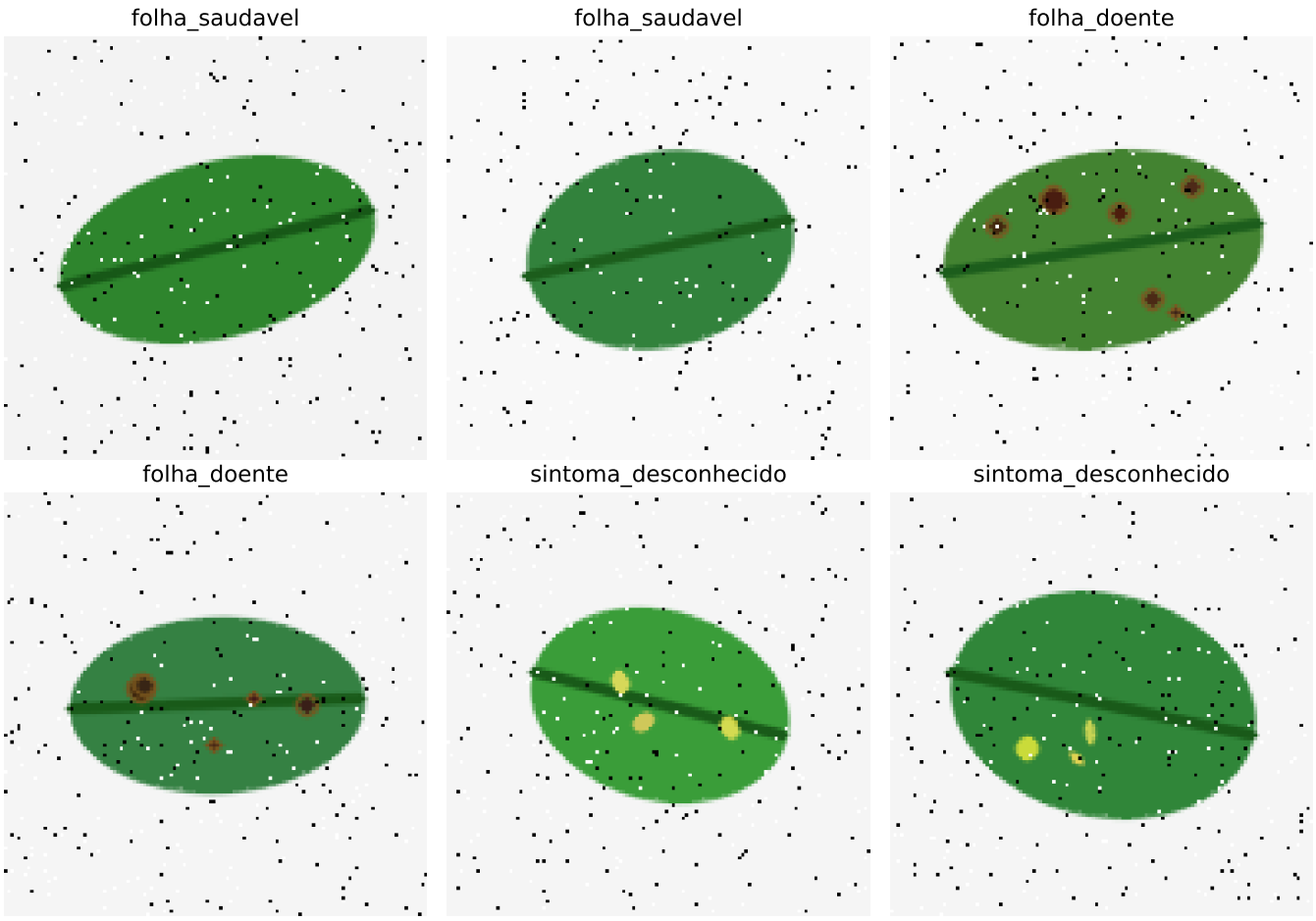


Figure 10.26: Échantillons synthétiques du *dataset* de diagnostic foliaire : feuille saine, feuille malade (taches sombres) et symptôme inconnu (chlorose jaunâtre) avec bruit sel et poivre.

> **Piège Fréquent**
>
> Le transfert d'apprentissage exige que chaque classe présente des motifs visuels distincts. Répéter la même image avec des étiquettes différentes empêche la couche de classification d'apprendre une frontière de décision, car l'extracteur génère pratiquement les mêmes caractéristiques pour tous les échantillons. Dans cette expérience, chaque image est générée de manière indépendante, avec des motifs visuels compatibles avec sa classe (feuille saine, lésions fongiques ou chlorose), fournissant ainsi des informations suffisantes pour l'entraînement de la couche de classification.

Bloco 2: Préparation des tenseurs et adaptation de l'architecture

Des modèles comme *ResNet-18* exigent des images colorées de 224×224 *pixels* normalisées selon les statistiques d'*ImageNet* ($\mu = [0,485; 0,456; 0,406]$ et $\sigma = [0,229; 0,224; 0,225]$).

1. **Transformation d'entrée (`transforms.Compose`):** on applique le redimensionnement et la normalisation standard à chaque image du *dataset* synthétique, produisant les tenseurs `X_treino`/`X_val` et les étiquettes `y_treino`/`y_val` — chaque exemple est une image réellement distincte, associée à l'étiquette correcte de sa classe.
2. **Gel de l'extracteur:** la boucle `for p in modelo_resnet.parameters(): p.requires_grad = False` désactive les gradients dans les couches convolutionnelles pré-entraînées.
3. **Nouvelle couche finale:** la couche `modelo_resnet.fc` est remplacée par une nouvelle instance `nn.Linear(modelo_resnet.fc.in_features, n_classes_destino)`, récemment initialisée et avec des gradients actifs par défaut. Le nombre de caractéristiques d'entrée est obtenu dynamiquement à

partir de la couche originale elle-même (`in_features`, égal à 512 dans ResNet-18), plutôt que d'être fixé manuellement dans le code — pratique recommandée, car elle rend le fragment réutilisable pour d'autres variantes de l'architecture sans modifications.

```
# 1. Pipeline de transformations attendues par ResNet
transformacao_resnet = T.Compose([
    T.Resize((224, 224)),
    T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])


def prepara_tensores(imgs, labels):
    tensores = [transformacao_resnet(T.functional.to_tensor(img)) for img in imgs]
    X = torch.stack(tensores)
    y = torch.tensor(labels, dtype=torch.long)
    return X, y


# 2. Conversion du dataset synthétique (Bloc 1) en tenseurs normalisés
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. Chargement de ResNet-18 pré-entraînée et gel de l'extracteur
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. Remplacement de la couche finale pour les 3 nouvelles classes cibles
modelo_resnet.fc = nn.Linear(modelo_resnet.fc.in_features, n_classes_destino)

print(f"Nouvelle couche finale : {modelo_resnet.fc}")
```

Bloco 3: Boucle d'Entraînement et Évaluation

Avec l'extracteur figé et la nouvelle couche de sortie correctement intégrée, on exécute l'ajustement fin (*fine-tuning*) de la nouvelle tête de classification.

1. **Optimisation ciblée :** l'optimiseur *Adam* reçoit strictement `modelo_resnet.fc.parameters()`, ne mettant à jour que la nouvelle couche de sortie — le reste du réseau reste figé, comme défini dans le Bloco 2.
2. **Exécution de la boucle :** à chaque époque, le modèle itère sur les lots d'entraînement, calcule la perte par entropie croisée et ajuste les poids de la couche finale ; ensuite, on évalue la précision sur l'ensemble de validation (images jamais vues durant l'entraînement).
3. **Courbes d'entraînement :** la Figure 10.27 suit l'évolution de la perte d'entraînement et de la précision de validation au fil des époques — comme les trois classes sont visuellement distinctes entre elles, on s'attend à une convergence réelle, bien au-dessus du seuil de 33% correspondant à un choix aléatoire parmi 3 classes.

Bloco 4: Inspeção Qualitativa das Previsões

Além da curva de precisão agregada, é instrutivo observar individualmente algumas previsões du modelo no conjunto de validação, comparando o rótulo previsto com o rótulo real. A Figure 10.28 exibe duas amostras de cada classe.

```
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"Époque {epoca+1}/{epocas} - perte : {historico_perda[-1]:.4f} - ",
          f" précision_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",
)
```

Figure 10.27

```python
modelo_resnet.eval()


# Deux échantillons de chaque classe dans l'ensemble de validation
idx_amostras = [0, N_POR_CLASSE_VAL, 2 * N_POR_CLASSE_VAL,
                1, N_POR_CLASSE_VAL + 1, 2 * N_POR_CLASSE_VAL + 1]

imgs_pred, titulos_pred = [], []
with torch.no_grad():
    for idx in idx_amostras:
        entrada = X_val[idx].unsqueeze(0).to(dispositivo)
        pred_idx = modelo_resnet(entrada).argmax(dim=1).item()
        real_idx = y_val[idx].item()
        marcador = " " if pred_idx == real_idx else " "
        imgs_pred.append(imgs_val[idx])
        titulos_pred.append(f"{marcador} previsto: {CLASSES_FOLHA[pred_idx]}\n"+
                            f"real: {CLASSES_FOLHA[real_idx]}")

mm.show(imgs_pred, titles=titulos_pred, cols=3, figsize=(10, 7))
```

Figure 10.28

Analyse de l'expérience 3

La *ResNet-18* atteint une haute précision même en utilisant un ensemble réduit d'images synthétiques. Ce résultat montre que les représentations apprises sur **ImageNet** restent utiles dans un domaine complètement différent, ne nécessitant que l'adaptation de la couche de classification.

L'expérience illustre également une situation courante dans les applications réelles, où la disponibilité de données étiquetées est limitée. Dans ces scénarios, le transfert d'apprentissage réduit le temps d'entraînement et permet d'obtenir des modèles performants même sans entraîner l'ensemble du réseau.

> 🧠 Synthèse comparative — Quand le transfert d'apprentissage fonctionne-t-il ?
>
> Les trois expériences montrent que le transfert d'apprentissage dépend de la capacité de généralisation de l'extracteur de caractéristiques.
>
> - **Expérience 1 :** un petit extracteur, entraîné sur un domaine restreint, apprend des représentations peu généralisables et peut produire un **transfert négatif**.
> - **Expérience 2 :** une *ResNet-18* pré-entraînée sur **ImageNet** transfère des représentations générales à une tâche de classification de races de chiens et de chats, atteignant une haute précision avec peu d'échantillons.
> - **Expérience 3 :** la même stratégie adapte le modèle à un problème de diagnostic phytosanitaire, montrant qu'un seul extracteur peut servir de base à différents domaines d'application.

Comparaison des Approches de Transfert

La Table 10.2 résume les résultats obtenus lors des trois expériences.

Tableau 10.2: Comparaison entre les trois scénarios de transfert d'apprentissage présentés dans cette section.

Aspect	Expérience 1	Expérience 2	Expérience 3
Extracteur	Petit CNN	*ResNet-18*	*ResNet-18*
Entraînement de l'extracteur	Chiffres (0–4)	**ImageNet**	**ImageNet**
Capacité de généralisation	Faible	Élevée	Élevée
Nouvelle tâche	Chiffres (5–9)	Races de chiens et de chats	Diagnostic phytosanitaire
Résultat	Transfert négatif	Transfert positif	Transfert positif

10.5.2 Détection d'objets

Les expériences précédentes ont montré comment le transfert d'apprentissage adapte des modèles pré-entraînés à des tâches de classification d'images. Le même principe sous-tend également les architectures de détection d'objets, où un extracteur de caractéristiques pré-entraîné fournit des représentations visuelles générales, tandis que des modules spécialisés localisent et classifient les objets dans l'image.

Les sections suivantes présentent la **Faster R-CNN** comme exemple de détecteur pré-entraîné utilisé directement pour l'inférence, puis une expérience complète d'ajustement fin avec l'architecture **YOLO**.

10.5.2.1 Faster R-CNN : Détecteur pré-entraîné

La détection d'objets étend l'utilisation de modèles pré-entraînés à une tâche plus complexe que la classification. La **Faster R-CNN** utilise une CNN pré-entraînée, comme la *ResNet-50*, comme **extracteur de caractéristiques** (*backbone*) et ajoute des modules spécialisés pour localiser et classifier les objets.

Concernant cet extracteur, l'architecture intègre deux « têtes » principales :

- ***Region Proposal Network* (RPN) :** propose des régions de l'image ayant une forte probabilité de contenir des objets.
- **Tête de classification :** affine ces régions, attribue une classe à chaque objet et ajuste ses boîtes englobantes.

Le code suivant utilise une Faster R-CNN avec des poids pré-entraînés sur **COCO** pour détecter des objets dans une image, produisant leurs classes, coordonnées et scores de confiance.

> ℹ 🔍 Où se trouve le transfert d'apprentissage ici ?
>
> Contrairement aux expériences précédentes, cet exemple **n'effectue pas d'ajustement fin** (*fine-tuning*). Le modèle exécute uniquement l'inférence (`eval()`), réutilisant directement les poids du *backbone*, de la RPN et de la tête de classification entraînés sur **COCO**.
> L'adaptation à un nouveau domaine exigerait de remplacer la couche `box_predictor` par une nouvelle tête de classification, compatible avec les classes de l'application, et de l'entraîner sur un ensemble d'images annotées. Cette procédure suit le même principe présenté dans la section sur le transfert d'apprentissage et constitue le flux habituel pour des applications spécifiques, telles que la détection de nuisibles, de défauts de fabrication ou de véhicules.

1. **Catégories COCO :** Le code récupère les noms des classes à partir des méta-informations des poids (`FasterRCNN_ResNet50_FPN_Weights.DEFAULT.meta["categories"]`). Bien que cette liste contienne 91 entrées pour des raisons historiques liées au format d'annotation de COCO, seules 80 correspondent à des catégories d'objets.

2. **Inférence :** L'image chargée par `mm.read()` est convertie en *tensor* et traitée par le modèle en mode évaluation (`eval()`). Le code ne conserve que les détections dont la confiance est supérieure à 80%.

3. **Annotation de l'image :** Pour chaque objet détecté, le code dessine la boîte englobante (`cv2.rectangle`) et écrit la classe prédite ainsi que sa confiance (`cv2.putText`).

4. **Visualisation :** La Figure 10.29 présente l'image annotée avec les détections réalisées par le modèle.

10.5.2.2 Détection d'objets et transfert d'apprentissage avec YOLO

Les sections précédentes ont appliqué le transfert d'apprentissage à des problèmes de **classification d'images**, où le modèle associe une seule étiquette à l'image entière. Dans cette section, le même principe est étendu à la **détection d'objets**, une tâche qui exige d'identifier simultanément **ce qui** est présent dans l'image et **où** se trouve chaque objet.

La sous-section précédente a présenté la **Faster R-CNN** comme exemple de détecteur pré-entraîné utilisé directement pour l'inférence, sans aucune adaptation au nouveau domaine. Dans cette expérience, le modèle passe par une étape d'**ajustement fin** (*fine-tuning*) : on part d'une architecture **YOLO** (*You Only Look Once*) pré-entraînée sur l'ensemble **COCO** et on adapte le réseau pour détecter et classer les objets d'un nouveau domaine.

Contrairement à la Faster R-CNN, qui effectue la détection en deux étapes, la famille **YOLO** adopte une architecture à étape unique (*single-stage detector*), estimant, en une seule propagation à travers le réseau, les boîtes englobantes (*bounding boxes*), la confiance de chaque détection et la classe correspondante. Cette stratégie réduit le coût computationnel et rend possibles des applications en temps réel.

À titre d'exemple, l'expérience utilise un ensemble synthétique de formes géométriques (triangles, carrés, étoiles, entre autres), avec des variations de couleur, de taille, de rotation et une dégradation par bruit de type sel et poivre.

Bloc 1 : Génération du *dataset* synthétique

Le bloc suivant génère un ensemble synthétique pour l'entraînement et l'évaluation du détecteur. Chaque image contient entre un et trois objets appartenant à l'une des neuf classes :

```
CLASSES = [
    'Triangle', 'Square', 'Pentagon', 'Hexagon',
    'Heptagon', 'Circle', 'Ellipse', 'Star', 'Cross'
]
```

1. **Génération des formes :** Les fonctions `poligono_regular`, `poligono_estrela` et `poligono_cruz` construisent les coordonnées des objets. La fonction `desenha_objeto` dessine chaque forme avec une position, une taille, une orientation et une couleur aléatoires et calcule sa *bounding box*.

2. **Annotation au format YOLO :** La fonction `gera_imagem_ruidosa` génère entre un et trois objets par image et convertit chaque *bounding box* au format YOLO, représenté par la classe et les coordonnées normalisées du centre, de la largeur et de la hauteur.

3. **Dégradation de l'image :** La fonction `adiciona_ruido_sal_pimenta` ajoute un bruit impulsif, simulant des imperfections d'acquisition.

4. **Visualisation des échantillons :** La Figure 10.30 présente des exemples de l'ensemble synthétique avec les *bounding boxes* superposées par la fonction `mm.showBoundBox()`.

```
CLASSES = [
    'Triangle', 'Square', 'Pentagon', 'Hexagon',
    'Heptagon', 'Circle', 'Ellipse', 'Star', 'Cross'
]
N_LADOS = {'Triangle': 3, 'Square': 4, 'Pentagon': 5, 'Hexagon': 6, 'Heptagon': 7}

def poligono_regular(cx, cy, r, n_lados, rot_graus):
    ang0 = np.deg2rad(rot_graus - 90)
```

```python
# 1. Chargement de l'image et des noms des catégories du 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"]  # Mappage indice -> nom de la classe

# 2. Chargement du modèle Faster R-CNN pré-entraîné
modelo_detection = fasterrcnn_resnet50_fpn(weights=pesos_coco).eval()

# 3. Exécution de l'inférence sans calcul de gradients
with torch.no_grad():
    predicao = modelo_detection([to_tensor(img)])[0]

# 4. Filtrage des détections avec une confiance supérieure à 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. Dessin des boîtes englobantes et des étiquettes de classe
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}"

    # Dessine le rectangle rouge (RGB : 255, 0, 0) avec une épaisseur de 3 pixels
    cv2.rectangle(img_com_caixas, (x1, y1), (x2, y2), (255, 0, 0), 3)

    # Écrit la classe et la confiance au-dessus de la boîte englobante
    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. Affichage graphique de l'image résultante
mm.show(img_com_caixas, title="Faster R-CNN (COCO) - Détection avec classe et confiance")
```

Figure 10.29

```
    angs = ang0 + 2 * np.pi * np.arange(n_lados) / n_lados
    return np.stack([cx + r * np.cos(angs), cy + r * np.sin(angs)], axis=1)

def poligono_estrela(cx, cy, r_externo, rot_graus, n_pontas=5):
    r_interno = r_externo * 0.45
    ang0 = np.deg2rad(rot_graus - 90)
    angs = ang0 + np.pi * np.arange(2 * n_pontas) / n_pontas
    raios = np.where(np.arange(2 * n_pontas) % 2 == 0, r_externo, r_interno)
    return np.stack([cx + raios * np.cos(angs), cy + raios * np.sin(angs)], axis=1)

def poligono_cruz(cx, cy, r, rot_graus, espessura_rel=0.35):
    w = r * espessura_rel
    base = np.array([
        (-w, -r), (w, -r), (w, -w), (r, -w), (r, w), (w, w),
        (w, r), (-w, r), (-w, w), (-r, w), (-r, -w), (-w, -w),
    ])
    theta = np.deg2rad(rot_graus)
    R = np.array([[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.cos(theta)]])
    return base @ R.T + np.array([cx, cy])

def desenha_objeto(img, classe_idx, cx, cy, tamanho, rotacao, cor):
    nome = CLASSES[classe_idx]
    if nome in N_LADOS:
        pts = poligono_regular(cx, cy, tamanho, N_LADOS[nome], rotacao)
        cv2.fillPoly(img, [pts.astype(np.int32)], cor)
        xs, ys = pts[:, 0], pts[:, 1]
    elif nome == 'Star':
        pts = poligono_estrela(cx, cy, tamanho, rotacao)
        cv2.fillPoly(img, [pts.astype(np.int32)], cor)
        xs, ys = pts[:, 0], pts[:, 1]
    elif nome == 'Cross':
        pts = poligono_cruz(cx, cy, tamanho, rotacao)
        cv2.fillPoly(img, [pts.astype(np.int32)], cor)
        xs, ys = pts[:, 0], pts[:, 1]
    elif nome == 'Circle':
        cv2.circle(img, (int(cx), int(cy)), int(tamanho), cor, -1)
        xs, ys = np.array([cx - tamanho, cx + tamanho]), np.array([cy - tamanho, cy +
tamanho])
    else:  # Ellipse
        eixo = (int(tamanho), int(tamanho * 0.6))
        cv2.ellipse(img, (int(cx), int(cy)), eixo, rotacao, 0, 360, cor, -1)
        ang = np.deg2rad(rotacao)
        dx = np.hypot(eixo[0] * np.cos(ang), eixo[1] * np.sin(ang))
        dy = np.hypot(eixo[0] * np.sin(ang), eixo[1] * np.cos(ang))
        xs, ys = np.array([cx - dx, cx + dx]), np.array([cy - dy, cy + dy])
    return xs.min(), ys.min(), xs.max(), ys.max()

def adiciona_ruido_sal_pimenta(img, quantidade=0.05):
    img_ruidosa = img.copy()
    h, w, c = img_ruidosa.shape
    num_ruido = int(quantidade * h * w)

    # Sel (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]

    # Poivre (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
```

```python
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)

        # Format YOLO : (classe, x_centre, y_centre, largeur, hauteur) normalisés
        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
```

```python
# Génération de 5 échantillons pour affichage initial en haut du projet
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)

    # Enregistrement temporaire de l'annotation pour lecture native par 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}")

# Affichage du panneau de 5 échantillons
mm.show(
    imgs_demo,
    titles=titulos_demo,
    cols=n_amostras_iniciais,
    figsize=(14, 3)
)
```

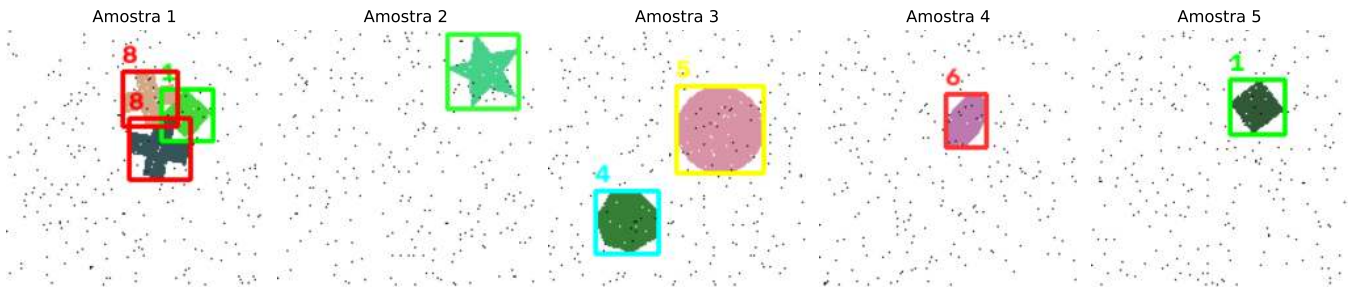


Figure 10.30: Échantillons initiaux du *dataset* synthétique bruité d'objets géométriques avec les boîtes englobantes du format YOLO superposées.

Bloc 2 : Organisation du *Dataset* et création du fichier `data.yaml`

Ce bloc organise l'ensemble de données au format attendu par la bibliothèque **Ultralytics YOLO**. Les images et les annotations sont réparties dans des répertoires séparés pour l'entraînement et la validation, tandis que le fichier `data.yaml` regroupe les informations nécessaires à l'entraînement du détecteur.

```
shapes_dataset/
   data.yaml
   images/
       train/
       val/
   labels/
        train/
        val/
```

1. **Génération de l'ensemble de données :** Le code crée **90** images pour l'entraînement et **20** pour la validation. Pour chaque image, il enregistre un fichier `.txt` contenant une ligne par objet, au format YOLO (`classe`, `x_c`, `y_c`, `largeur`, `hauteur`), avec toutes les coordonnées normalisées.
2. **Organisation des fichiers :** Les images sont stockées dans `images/train` et `images/val`, tandis que les annotations correspondantes sont enregistrées dans `labels/train` et `labels/val`, en préservant le même nom de fichier.
3. **Création du fichier `data.yaml` :** Le code génère automatiquement le fichier de configuration contenant le chemin du *dataset*, les répertoires d'entraînement et de validation, ainsi que la correspondance entre les indices numériques et les noms des neuf classes.

```
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("Jeu de données bruité généré avec succès : 90 images d'entraînement et 20 de
validation.")
```

```
Jeu de données bruité généré avec succès : 90 images d'entraînement et 20 de validation.
```

Bloco 3: Pré-traitement avec filtre médian

Ce bloc applique un pré-traitement pour réduire l'effet du bruit de type sel et poivre introduit lors de la génération du *dataset*. Le **filtre médian** (`cv2.medianBlur`) supprime ce type de dégradation tout en préservant mieux les contours des objets que les filtres de lissage conventionnels.

1. **Filtrage de l'image :** Le code applique un filtre médian avec une fenêtre 3×3 à l'image bruitée, réduisant les *pixels* impulsifs sans modifier les annotations de l'ensemble de données.

2. **Visualisation comparative :** La Figure 10.31 compare l'image originale et l'image filtrée, en maintenant les *bounding boxes* superposées grâce à la fonction `mm.showBoundBox()`.

```
# 1. Chargement du premier échantillon bruité du 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. Prétraitement avec Filtre Médian (fenêtre 3x3)
img_filtrada = cv2.medianBlur(img_ruidosa, ksize=3)

# 3. Superposition des bounding boxes avec 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. Affichage comparatif avec mm.show
mm.show(
    [img_ruid_anotada, img_filt_anotada],
    titles=[
        "1. Image Bruyante Originale (Sel et Poivre)",
        "2. Prétraitée (Filtre Médian 3x3)"
    ],
    cols=2,
    figsize=(9, 4)
)
```

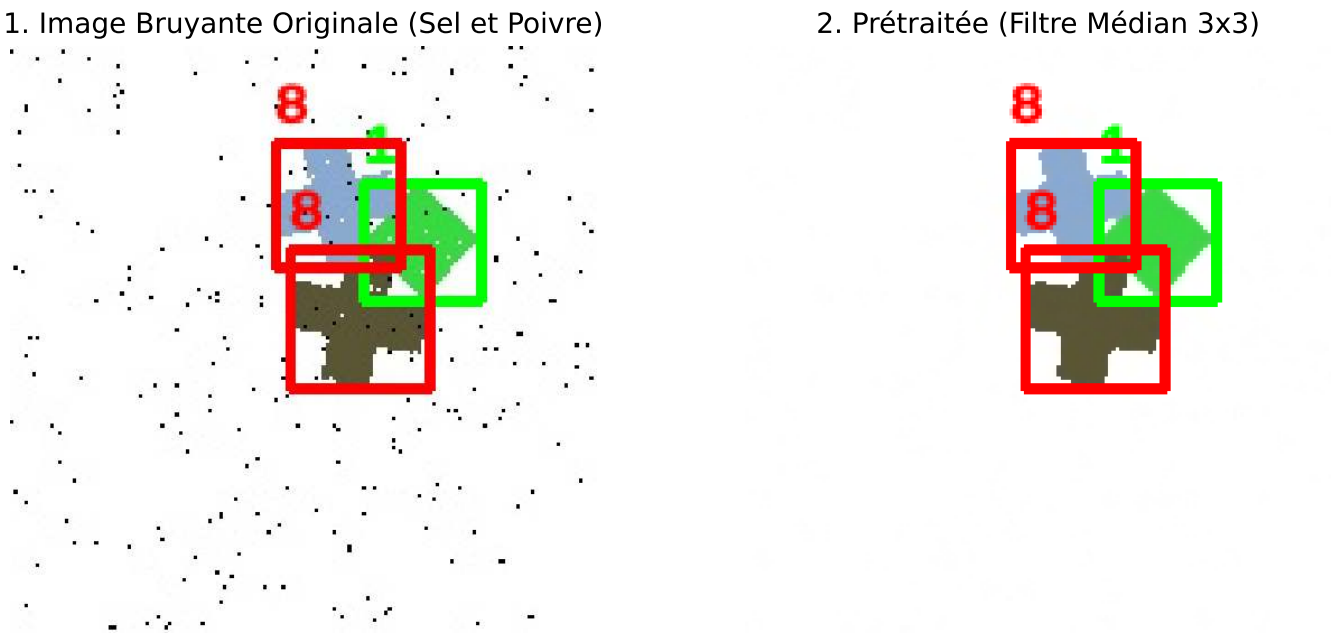


Figure 10.31: Comparaison entre l'image originale avec bruit de type sel et poivre et l'image après application du Filtre Médian (3x3). Les *bounding boxes* au format YOLO restent inchangées.

Bloco 4: Prétraitement du *dataset* et ajustement fin de YOLOv8

Ce bloc applique le filtre médian à l'ensemble d'images et réalise l'ajustement fin (*fine-tuning*) du détecteur **YOLOv8n** pré-entraîné sur l'ensemble **COCO**. Le filtrage réduit l'effet du bruit impulsif introduit lors de la génération des images, tandis que l'entraînement adapte les paramètres du réseau au nouveau domaine de formes géométriques.

1. **Filtrage par lots :** Le code parcourt les répertoires `train` et `val` et applique `cv2.medianBlur` avec une fenêtre 3×3 sur toutes les images, en conservant les annotations YOLO d'origine.
2. **Ajustement fin du détecteur :** Le réseau `YOLO("yolov8n.pt")`, initialement entraîné sur COCO, est adapté à l'ensemble géométrique via la fonction `.train()`. L'entraînement utilise des images avec une résolution de 320×320 *pixels* pendant 30 époques.
3. **Évaluation du modèle :** La fonction `.val()` calcule les métriques de détection sur l'ensemble de validation, y compris la **précision** (*precision*), le **rappel** (*recall*) et la **mAP50** (*mean Average Precision* avec un seuil d'IoU égal à $0,5$).

```
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

base_dir = "shapes_dataset"
base_dir_filt = "shapes_dataset_filtrado"

# 1. Crée une COPIE filtrée du jeu de données dans un dossier séparé
#    (le jeu de données original dans base_dir reste bruité, intact)
for split in ["train", "val"]:
    pasta_imgs_orig = f"{base_dir}/images/{split}"
    pasta_labels_orig = f"{base_dir}/labels/{split}"
    pasta_imgs_filt = f"{base_dir_filt}/images/{split}"
    pasta_labels_filt = f"{base_dir_filt}/labels/{split}"

    os.makedirs(pasta_imgs_filt, exist_ok=True)
```

```
    os.makedirs(pasta_labels_filt, exist_ok=True)

    for nome_arq in os.listdir(pasta_imgs_orig):
        if nome_arq.endswith(".jpg"):
            img_ruidosa = cv2.imread(f"{pasta_imgs_orig}/{nome_arq}")
            img_filtrada = cv2.medianBlur(img_ruidosa, ksize=3)
            # écrit la version filtrée dans le dossier NOUVEAU, ne remplace pas l'original
            cv2.imwrite(f"{pasta_imgs_filt}/{nome_arq}", img_filtrada)

    # copie les étiquettes (elles ne changent pas avec le filtre)
    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 pointant vers le jeu de données FILTRÉ
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("Prétraitement terminé : jeu de données filtré enregistré dans un dossier séparé.\n")

# Télécharger le modèle YOLOv8 pré-entraîné (yolov8n.pt) s'il n'est pas déjà présent
with contextlib.redirect_stdout(io.StringIO()), \
    contextlib.redirect_stderr(io.StringIO()):
    modelo_yolo = YOLO("yolov8n.pt")
print("Modèle YOLOv8 chargé.")
```

```
Prétraitement terminé : jeu de données filtré enregistré dans un dossier séparé.

Modèle YOLOv8 chargé.
```

```
# 3. Callback personnalisé pour imprimer uniquement l'époque en cours d'exécution
def on_train_epoch_start(trainer):
    epoch_atual = trainer.epoch + 1
    total_epochs = trainer.epochs
    # Écrit directement dans la sortie standard d'origine (en contournant le silencieux)
    sys.__stdout__.write(f"  Processando Época {epoch_atual}/{total_epochs}...\n")
    sys.__stdout__.flush()

# Ajoute le callback au modèle
modelo_yolo.add_callback("on_train_epoch_start", on_train_epoch_start)

# 4. Gestionnaire de contexte pour silencer les déchets d'Ultralytics (C/C++ et 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

# Exécution de l'entraînement
print("--- Démarrage de l'entraînement YOLOv8 ---")
with silenciar_logs():
    resultados_treino = modelo_yolo.train(
        data=f"{base_dir_filt}/data.yaml",   # <-- entraîne sur le jeu de données filtré
        epochs=30,
        imgsz=320,
        batch=16,
        device=device,
        verbose=False,
        plots=False
    )
    metricas = modelo_yolo.val(verbose=False)

# 5. Métriques finales
precision = metricas.results_dict["metrics/precision(B)"]
recall = metricas.results_dict["metrics/recall(B)"]
map50 = metricas.results_dict["metrics/mAP50(B)"]

print("\n--- Performance optimisée du modèle YOLOv8 ---")
print(f"Précision : {precision*100:.2f}%")
print(f"Rappel (Recall) : {recall*100:.2f}%")
print(f"mAP à 50% (IoU 0,50) : {map50*100:.2f}%")
```

```
--- Démarrage de l'entraînement YOLOv8 ---

--- Performance optimisée du modèle YOLOv8 ---
Précision : 70.98%
Rappel (Recall) : 89.05%
mAP à 50% (IoU 0,50) : 88.26%
```

Bloc 5 : Comparaison d'inférence : Image bruitée et image restaurée

Après le réglage fin de YOLOv8 sur l'ensemble restauré, une comparaison visuelle est effectuée entre la détection appliquée directement sur une image dégradée par le bruit sal et poivre et la même image après le filtre médian.

L'objectif est d'observer comment une étape simple de prétraitement peut influencer la qualité des prédictions d'un détecteur déjà adapté au nouveau domaine.

1. **Inférence avec YOLO :** La fonction `modelo_yolo.predict()` exécute la détection sur les deux versions de l'image, en utilisant un seuil de confiance de 25% (`conf=0.25`).
2. **Visualisation des prédictions :** La fonction `plot()` génère les images annotées avec les boîtes englobantes et les étiquettes prédites par le modèle. La Figure 10.32 présente la comparaison entre les deux scénarios.

```
# Nettoyage explicite des données téléchargées et libération de la mémoire
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))
```

```
# 1. Chargement d'un échantillon de test original (sans le filtre sauvegardé en lot)
caminho_teste = f"{base_dir}/images/val/0002.jpg"
img_ruidosa_teste = mm.read(caminho_teste)

# 2. Application ponctuelle du Filtre Médian (3x3) pour comparaison
img_filtrada_teste = cv2.medianBlur(img_ruidosa_teste, ksize=3)


# 3. Inférence avec le modèle YOLOv8 entraîné
pred_ruidosa = modelo_yolo.predict(img_ruidosa_teste, conf=0.25, verbose=False)[0]
pred_filtrada = modelo_yolo.predict(img_filtrada_teste, conf=0.25, verbose=False)[0]

# 4. Extraction des matrices annotées par le générateur du YOLO (conversion 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. Affichage comparatif standardisé via mm.show
mm.show(
    [img_pred_ruid, img_pred_filt],
    titles=[
        f"Inférence sur l'image bruitée ({len(pred_ruidosa.boxes)} objets)",
        f"Inférence sur l'image filtrée ({len(pred_filtrada.boxes)} objets)"
    ],
    cols=2,
    figsize=(10, 4)
)
```

Figure 10.32

```
    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 domaine bien plus éloigné que celui des chiffres

Dans l'expérience de transfert d'apprentissage entre chiffres manuscrits (domaines A et B), la tâche source et la tâche cible partageaient des statistiques visuelles très proches : toutes deux étaient des traits en niveaux de gris sur fond uniforme. Ici, la distance entre les domaines est bien plus grande — le YOLO a été pré-entraîné sur des **photographies naturelles en couleur** de COCO (personnes, animaux, véhicules, objets du quotidien), et la tâche cible consiste en **formes géométriques synthétiques, de couleur unie et au contour bien défini**, sans texture, éclairage ni arrière-plan complexe.
Néanmoins, le transfert d'apprentissage reste avantageux : les couches initiales d'un détecteur entraîné sur COCO apprennent des filtres génériques — détecteurs de bords, de coins et de régions de contraste — qui restent utiles pour délimiter le contour d'un triangle ou d'une étoile, même si le contenu visuel final est assez distinct. C'est pourquoi permettre l'ajustement fin (*fine-tuning*) de toutes les couches, combiné à un prétraitement cohérent entre l'entraînement et l'inférence pour atténuer le bruit sel et poivre, permet d'atteindre des taux de précision élevés dans la détection d'objets du nouveau domaine.

Utilisation d'un Ensemble de Données Réel

Le *pipeline* ci-dessus a été entièrement construit autour du **format d'annotation YOLO** (classe, x_{centre}, y_{centre}, largeur, hauteur, normalisés par la largeur et la hauteur de l'image), précisément afin qu'il puisse être réutilisé sans modification si le lecteur dispose d'un ensemble d'images réelles annotées de la même manière — par exemple, un ensemble d'images d'objets géométriques photographiés ou rendus, chacun accompagné d'un fichier `.txt` correspondant au même format utilisé ici. Pour cela, il suffirait de :

1. Organiser les images réelles dans `shapes_dataset/images/train` et `shapes_dataset/images/val`, ainsi que les fichiers `.txt` d'annotation correspondants dans les dossiers `labels/train` et `labels/val` (un fichier d'annotation par image, même nom de base, extension `.txt`) ;
2. Ajuster le fichier `data.yaml` si le nombre ou les noms des classes diffèrent ;
3. Exécuter les mêmes cellules d'entraînement, d'ajustement fin et de visualisation déjà présentées, sans aucune autre modification de code.

Cette séparation entre **génération/organisation des données** et **entraînement du modèle** est, en pratique, la raison pour laquelle les formats d'annotation standardisés (comme celui du YOLO) sont si largement adoptés : ils permettent de remplacer l'ensemble de données d'entrée — synthétique par réel, un domaine par un autre — tout en maintenant inchangé tout le reste du *pipeline* de transfert d'apprentissage.

10.5.3 Segmentation d'objets

Le même principe de transfert d'apprentissage sous-tend également les architectures de segmentation d'images, dans lesquelles un extracteur de caractéristiques pré-entraîné fournit des représentations visuelles générales, tandis qu'une tête spécialisée effectue la classification dense, pixel par pixel.

Les sections suivantes présentent **DeepLabV3** comme exemple de segmentateur pré-entraîné utilisé directement pour l'inférence, puis l'architecture **U-Net**, entraînée de zéro et comparée à une ligne de base morphologique classique.

10.5.3.1 DeepLabV3 : Segmentateur Pré-entraîné

En segmentation sémantique, l'objectif ne se limite pas à la localisation des objets par des boîtes englobantes (*bounding boxes*). Le réseau attribue une classe à chaque *pixel* de l'image, produisant une carte d'étiquettes ayant la même résolution que l'entrée. Des architectures telles que **DeepLabV3**, avec un *backbone* **ResNet-50**, utilisent un extracteur de caractéristiques pré-entraîné et une tête spécialisée pour réaliser cette classification dense.

1. **Chargement et inférence :** Le modèle `deeplabv3_resnet50(weights="DEFAULT")` charge des poids pré-entraînés sur l'ensemble **Pascal VOC**, qui définit 21 classes de segmentation. Le code convertit l'image en *tensor*, ajoute la dimension de *batch* (`unsqueeze(0)`) et exécute l'inférence.

2. **Carte des classes :** La sortie du modèle possède la dimension $(1, 21, H, W)$, contenant une valeur pour chaque classe à chaque *pixel*. L'opération `.argmax(dim=1)` sélectionne la classe ayant la réponse la plus élevée à chaque position, générant une matrice bidimensionnelle d'étiquettes de dimensions (H, W).

3. **Visualisation :** Le code convertit la carte d'étiquettes au format attendu par `mm.show()`, qui affiche le résultat de la segmentation sur la Figure 10.33..

10.5.3.2 Segmentação Semântica com Arquitetura U-Net

A subseção anterior apresentou um modelo pré-treinado (*DeepLabV3*) que produz diretamente um rótulo de classe por *pixel*. Esta seção completa a sequência de projetos práticos — classificação e detecção — abordando a segmentação semântica implementada e treinada do zero com a **U-Net**, a arquitetura de referência introduzida por Ronneberger (2015).

```
# 1. Chargement de l'image (retourne 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. Chargement du modèle DeepLabV3 pré-entraîné en mode évaluation
modelo_segmentacao = deeplabv3_resnet50(weights="DEFAULT").eval()

# 3. Exécution de l'inférence sans calcul de gradients

with torch.no_grad():
    tensor_entrada = to_tensor(img).unsqueeze(0)  # Format (1, C, H, W)
    saida = modelo_segmentacao(tensor_entrada)["out"]

    # Sélection de la classe avec la plus grande probabilité par pixel (argmax sur l'axe des
    canaux)
    mapa_classes = saida.argmax(dim=1).squeeze(0).byte().cpu().numpy()

# 4. Affichage de la carte de segmentation sémantique

mm.show(
    [img,mapa_classes],
    title=["Image originale","Segmentation sémantique (DeepLabV3)"]
)
```

Figure 10.33

Na classificação, o mapa de características final era achatado (*flatten*) em um vetor, descartando a informação espacial em favor de um único rótulo por imagem. A U-Net, por sua vez, produz uma saída com a **mesma resolução espacial da entrada**: um mapa bidimensional no qual cada *pixel* recebe sua própria classificação. Essa exigência — preservar detalhe espacial de alta resolução ao mesmo tempo em que se constrói contexto semântico em camadas profundas — motiva a arquitetura de codificador-decodificador com conexões de atalho (*skip connections*).

O cenário utilizado simula a segmentação de nódulos em exames médicos sintéticos: imagens em escala de cinza contêm uma região circular ("nódulo") sobreposta a um fundo, ambos contaminados por ruído gaussiano e com médias de intensidade muito próximas — um desafio intencional de baixo contraste, ideal para demonstrar o ganho do aprendizado espacial em relação à limiarização pontual.

Bloc 1 : Générateur de l'ensemble synthétique de nodules et affichage initial

Le générateur suivant produit des paires (image, masque) : l'image contient une région circulaire d'intensité légèrement supérieure à celle du fond, toutes deux affectées par le même écart-type de bruit gaussien. Le masque binaire délimite exactement la région du nodule et sert de vérité de référence (*ground truth*).

1. **Construction du nodule :** La fonction `gera_imagem_com_nodulo` superpose le nodule au fond sur une matrice 64×64 et applique un bruit gaussien.
2. **Visualisation avec `mm.show` :** La Figure 10.34 illustre les deux premiers échantillons et leurs masques respectifs.

```
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


# Génération des ensembles d'entraînement et de validation
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"Ensemble d'entraînement : {N_TREINO} images | Ensemble de validation : {N_VAL} images\n"
)

# Affichage d'échantillons initiaux
mm.show(
    [imgs_treino[0], masks_treino[0], imgs_treino[1], masks_treino[1]],
    titles=["Image 1", "Masque 1", "Image 2", "Masque 2"],
    cols=4,
    figsize=(11, 3),
)
```

```
Ensemble d'entraînement : 160 images | Ensemble de validation : 40 images
```

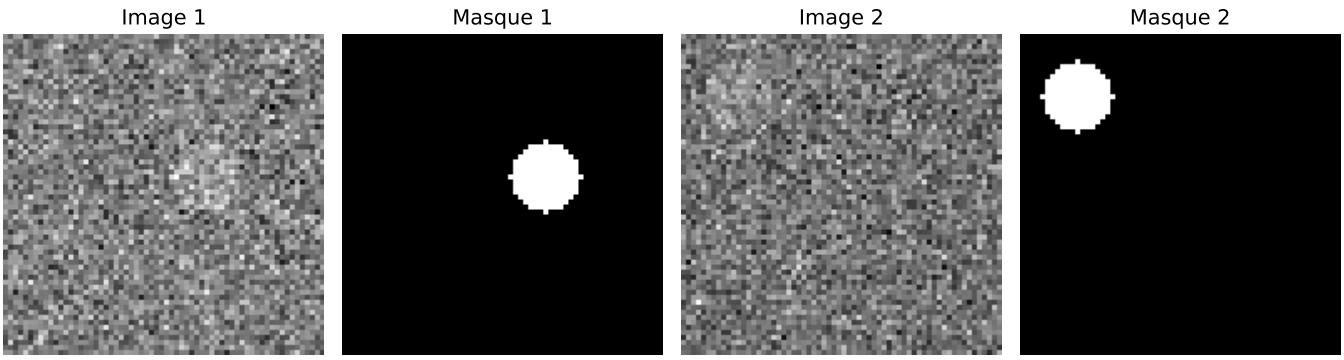


Figure 10.34: Échantillons du jeu de données synthétique de nodules : image en niveaux de gris sous bruit et masque binaire de référence correspondant.

Bloco 2: Ligne de base classique (filtrage, Otsu et morphologie)

Avant d'employer l'U-Net, on évalue la performance d'un *pipeline* morphologique classique construit avec la bibliothèque `morph` : un **lisseur gaussien** (`mm.blur`), un **seuillage d'Otsu** (`mm.threshold`) et une **ouverture morphologique** (`mm.open`) pour l'élimination des bruits isolés.

1. **Métrique d'Intersection sur Union (IoU) :** La fonction `iou_mascaras` calcule le degré de chevauchement *pixel par pixel* entre la prédiction et le masque réel.
2. **Exécution et comparatif :** La Figure 10.35 affiche le résultat de la segmentation classique sur une image de test, démontrant les limitations du seuil global sous faible 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 moyen (ligne de base classique) sur la validation : {iou_classico_medio:.4f}\n"
)

predicao_classica_exemplo = segmenta_classico(imgs_val[0], elemento_estrutural)

mm.show(
    [imgs_val[0], masks_val[0], predicao_classica_exemplo],
    titles=["Image", "Masque de référence", "Prédiction classique"],
    cols=3,
    figsize=(9, 3.2),
)
```

```
IoU moyen (ligne de base classique) sur la validation : 0.7516
```

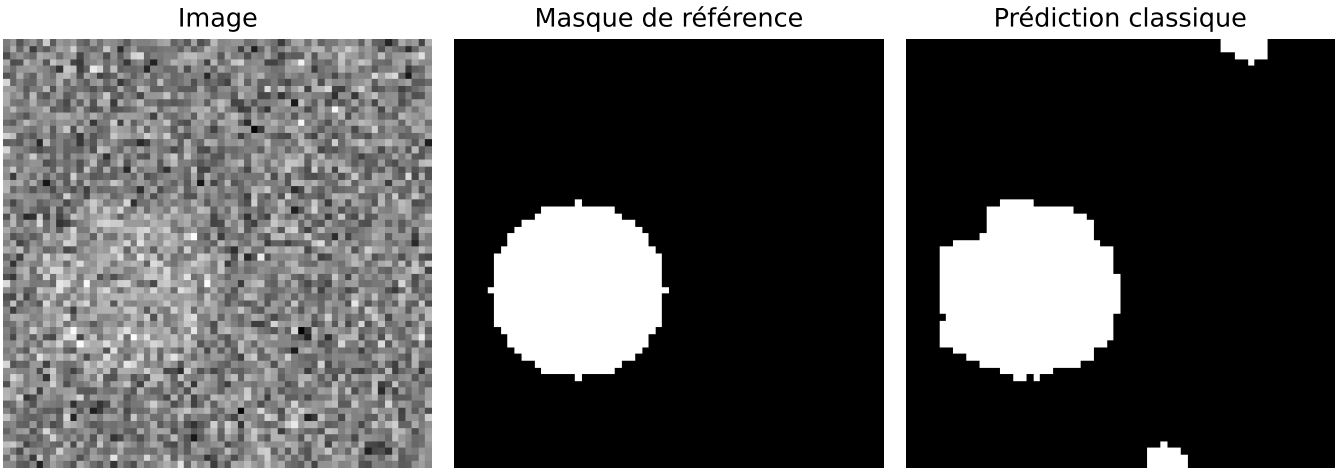


Figure 10.35: Ligne de base classique de segmentation : lissage, seuillage d'Otsu et ouverture morphologique affichés.

Bloco 3: Construction de l'architecture U-Net et fonctions de perte

La U-Net est une architecture en forme de « U » (d'où son nom), conçue spécifiquement pour la segmentation d'images. Elle est composée de deux chemins qui travaillent ensemble :

- ↓ **Encodeur :** descend dans l'image, réduisant la résolution spatiale à chaque étape tout en extrayant des caractéristiques de plus en plus abstraites (contours → textures → formes → contexte).
- **Décodeur :** remonte, reconstruisant la résolution originale grâce à des convolutions transposées (*upsampling*), jusqu'à générer un masque de la même taille que l'image d'entrée.

L'élément qui rend la U-Net spéciale, ce sont les **connexions de saut** (*skip connections*) : elles acheminent les cartes de caractéristiques de l'encodeur directement vers l'étape correspondante du décodeur, à la même résolution. Cela évite que des détails fins — comme les contours et les bords — ne se perdent lors de la compression spatiale.

Flux général de l'architecture :

```
Entrée


Encodeur (Conv → Conv → Pool) × 3

      connexions de saut réinjectant les cartes haute résolution

Base (goulot d'étranglement)


Décodeur (Upsample → Concat → Conv → Conv) × 3


Sortie 1×1 (logits)
```

Composants principaux :

1. **Bloc convolutif de base (`BlocoConv`)** L'unité fondamentale répétée dans tout le réseau. Elle applique deux convolutions 3×3 en séquence, chacune suivie d'une activation ReLU, avec un *padding* qui préserve les dimensions spatiales de l'entrée. C'est ce bloc qui apparaît aussi bien dans l'encodeur que dans le décodeur.

2. **Convolution transposée (`nn.ConvTranspose2d`)** C'est l'opération responsable de l'*upsampling* dans le décodeur : au lieu de réduire la résolution spatiale (comme le fait le `MaxPool2d` dans l'encodeur), elle

l'augmente, apprenant les poids nécessaires pour « défaire » la compression et retrouver progressivement la taille originale de l'image.

3. **Perte combinée (BCE + Dice)** La fonction `perda_segmentacao` additionne deux métriques complémentaires :

   - **Entropie croisée binaire (BCE) :** évalue la justesse *pixel par pixel*.
   - **Coefficient de Dice :** évalue la *superposition globale* entre le masque prédit et le masque réel.

   Ensemble, elles équilibrent la précision locale avec la fidélité de la forme segmentée dans son ensemble.

```
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)
```

Bloc 4 : Entraînement de la U-Net et Évaluation des Performances

L'entraînement est exécuté sur 35 époques en utilisant l'optimiseur *Adam*. À chaque époque, on surveille la Perte d'Entraînement et l'indice IoU moyen sur l'ensemble de validation.

1. **Boucle d'Entraînement :** Les paramètres sont mis à jour avec des *mini-lots* de 16 échantillons.
2. **Évolution Graphique :** La Figure 10.36 affiche le graphique avec la progression de la perte et de l'IoU.

```
modelo_unet = UNetCompacta()
print(
    f"Paramètres entraînables de l'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
epocas_unet = 35
historico_perda_unet, historico_iou_unet = [], []

for epoca in range(epocas_unet):
    if epoca % 5 == 0:  # Imprime toutes les 5 époques
        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 moyen final de l'U-Net en validation : {iou_unet_final:.4f}")

# Graphique de l'évolution de l'entraînement
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")
```

```
Paramètres entraînables de l'U-Net : 120681
1 / 35
6 / 35
11 / 35
16 / 35
21 / 35
26 / 35
31 / 35
IoU moyen final de l'U-Net en validation : 0.8876
```

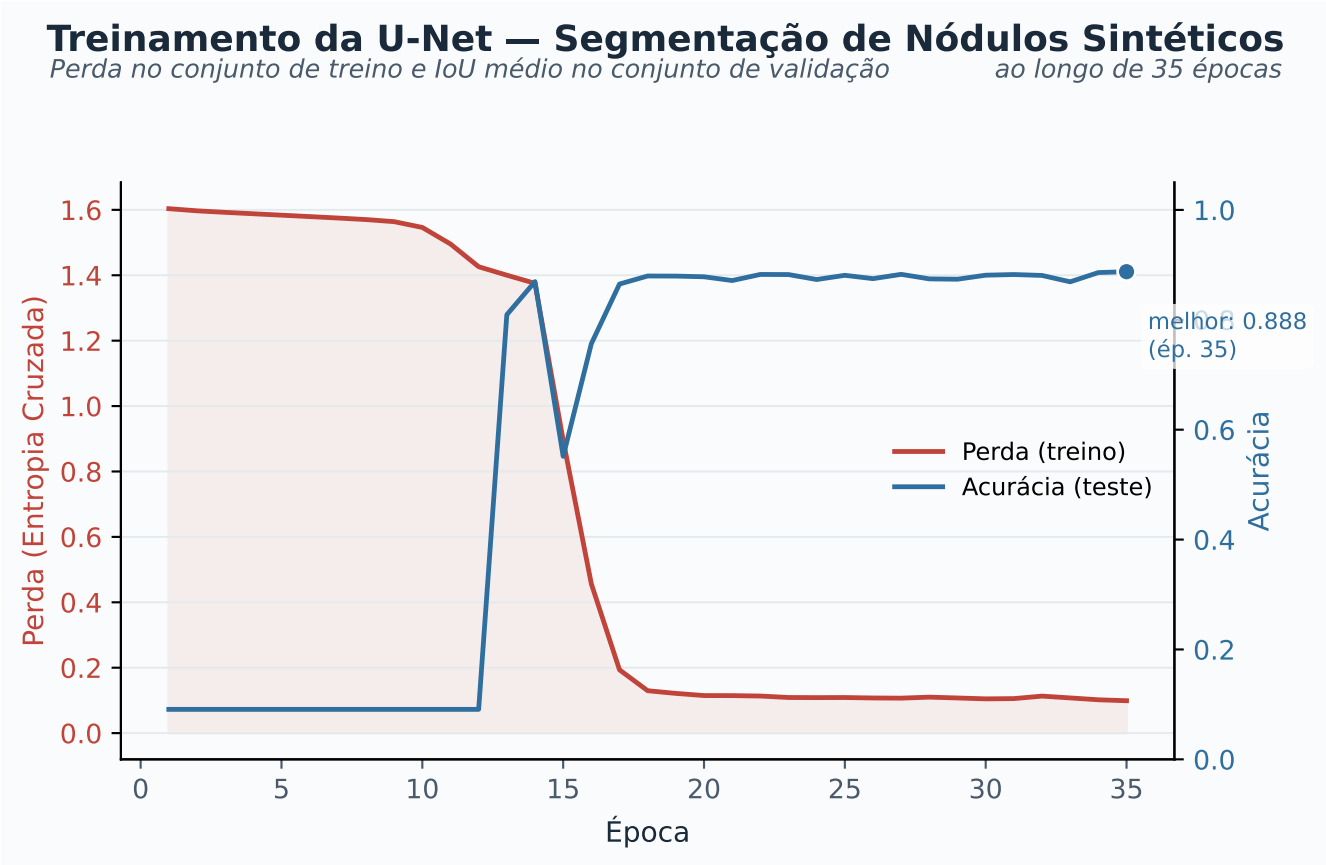


Figure 10.36: Évolution de l'entraînement de l'U-Net compacte : réduction de la perte et élévation de l'indice IoU moyen sur l'ensemble de validation au fil de 35 époques.

Bloc 5 : Comparatif Quantitatif et Qualitatif (Classique vs. U-Net)

La comparaison entre l'approche classique et la U-Net met en évidence la supériorité de l'apprentissage de représentations dans les scénarios à faible contraste.

1. **Tableau de métriques :** Le graphique à barres dans la Figure 10.37 contraste le IoU moyen des deux méthodes lors de la validation.
2. **Visualisation qualitative :** La comparaison visuelle sur trois échantillons démontre comment les connexions de raccourci permettent de récupérer le contour du nodule, même sous un bruit prononcé.

```
# 1.  comparatif
print(f"IoU moyen (Lissage classique + Otsu) : {iou_classico_medio:.4f}")
print(f"IoU moyen (U-Net) : {iou_unet_final:.4f}")

# 2. Affichage qualitatif côte à côte sur 3 échantillons via mm.show
imgs_comparativas = []
titulos_comparativos = []

for i in range(3):
    with torch.no_grad():
        pred_unet = (
            (torch.sigmoid(modelo_unet(X_val_unet[i : i + 1])) > 0.5)
            .float()
            .squeeze()
            .numpy()
            * 255
        )
```

```
    pred_classico = segmenta_classico(imgs_val[i], elemento_estrutural)

    imgs_comparativas.extend(
        [imgs_val[i], masks_val[i], pred_classico, pred_unet.astype(np.uint8)]
    )

    t_prefix = f"Amostra {i+1}"
    titulos_comparativos.extend(
        [
            f"{t_prefix}: Imagem",
            f"{t_prefix}: Referência",
            f"{t_prefix}: Clássico",
            f"{t_prefix}: U-Net",
        ]
    )

mm.show(
    imgs_comparativas,
    titles=titulos_comparativos,
    cols=4,
    figsize=(11, 7.5),
)
```

```
IoU moyen (Lissage classique + Otsu) : 0.7516
IoU moyen (U-Net) : 0.8876
```

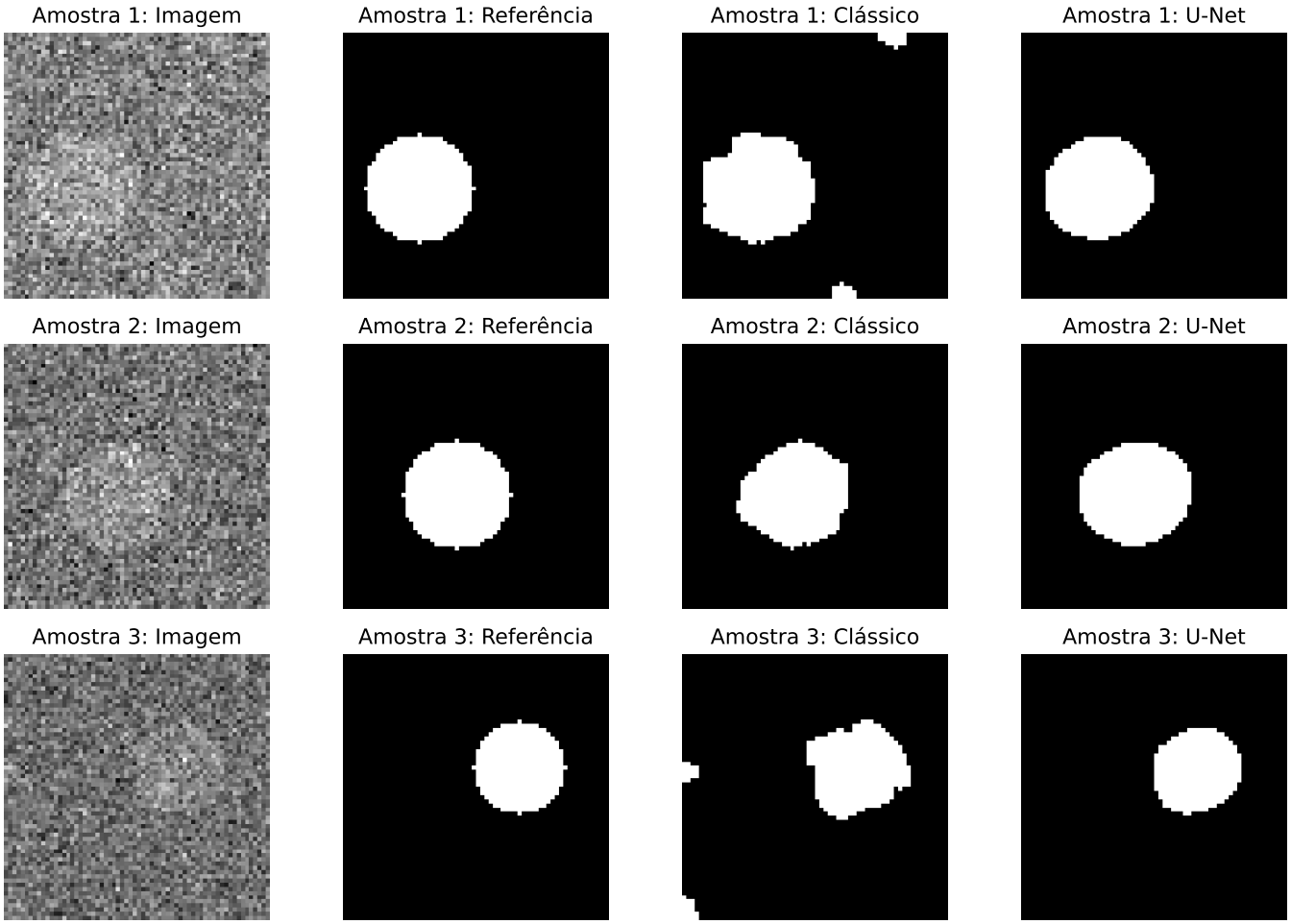


Figure 10.37: Comparativo quantitatif (IoU moyen) et qualitatif entre l'approche classique et l'U-Net entraînée sur trois échantillons de validation.

🧠 Pourquoi l'U-Net surpasse-t-elle le seuillage fixe ?

Le seuillage d'Otsu applique une valeur de coupure globale sur l'intensité locale. Lorsque la différence de moyenne entre le nodule et le fond est faible par rapport au bruit gaussien, cette règle commet des erreurs systématiques au niveau des bords.
L'U-Net contourne cette limitation en combinant le contexte sémantique large extrait par l'encodeur avec les détails spatiaux fins préservés par les connexions de saut. Cela permet d'identifier la présence du nodule et de délimiter ses contours avec précision, même sous un bruit intense.

10.5.4 Ingénierie des données pour la vision par ordinateur (*Roboflow*)

Une U-Net peut être entraînée à partir d'images annotées sur des plateformes d'ingénierie des données pour la vision par ordinateur, telles que *Roboflow*. Ces plateformes permettent d'organiser des ensembles de données, de réaliser des annotations, d'appliquer des étapes de prétraitement et d'augmentation des données, d'entraîner des modèles et d'exporter les données dans différents formats. Dans cette section, toutefois, l'exemple reprend la **détection d'objets géométriques**, en utilisant des ensembles de données déjà présentés dans cet ouvrage.

! Dépendance à la connexion et à la clé API

Les cellules de cette section exigent une connexion à Internet et une clé API gratuite de *Roboflow* (`app.roboflow.com`). Dans l'*espace de travail* du projet, accédez à **§ → API Roboflow** et copiez la **clé API privée**.
Créez, dans ce dossier, le fichier `chave_roboflow.txt` contenant **uniquement la clé**, sans guillemets. Un modèle de ce fichier est disponible dans `chave_roboflow.txt.exemplo`.
Ajoutez `chave_roboflow.txt` au fichier `.gitignore`, car il contient une information d'authentification qui ne doit être ni versionnée ni partagée.

10.5.4.1 Détection d'objets avec *Roboflow*

Roboflow permet d'effectuer une inférence sur des images locales, permettant d'évaluer les performances du modèle hébergé dans l'identification des objets d'intérêt.

Outre l'inférence, le *dataset* peut être exporté au format `png-mask-semantic`, dans lequel chaque image est accompagnée d'un masque de segmentation sémantique. Dans ce masque, chaque pixel représente la classe à laquelle appartient l'objet correspondant. Les paires image-masque sont utilisées comme données d'entraînement pour la `UNetCompacta`.

10.5.4.2 Connexion, *téléchargement* et vérification du *dataset*

Le code établit une connexion avec le *Workspace* `mctest` et le projet `geometric-test00`, version 6, puis effectue le *téléchargement* du *dataset* vers `dados/datasetRoboFlow`. Ensuite, il vérifie le *shape* des images dans chaque *split*. La fonction de vérification est réutilisée ultérieurement dans la section.

```
from roboflow import Roboflow
from pathlib import Path
from PIL import Image
from collections import Counter

def contar_shapes(raiz, splits=("train", "valid", "test")):
    """Compte la forme (hauteur, largeur, canaux) des images par split."""
    for split in splits:
        pasta = raiz / split / "images"
        if not pasta.exists():
            print(f"{split}: dossier introuvable")
```

```
            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("Clé Roboflow introuvable : clé_roboflow.txt")
else:
    with open(chave) as f:
        api_key = f.read().strip()

    # Projet :
    # 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} | Nom : {versao.name} | "
        f"Images : {versao.images}"
    )

    raiz = Path("dados/datasetRoboFlow")
    versao.download("yolov8", location=str(raiz))

    print("Dataset :", raiz)
    contar_shapes(raiz)
```

10.5.4.3 Téléchargement du *dataset* de manière reproductible (alternative)

En alternative au *dataset* obtenu via *Roboflow*, on peut utiliser un *dataset* disponible dans un dépôt GitHub, également organisé selon les mêmes *splits* et contenant les mêmes classes d'objets. Les ensembles de données ne sont toutefois pas identiques : les images du GitHub ont une résolution de `608×608` pixels, tandis que les images exportées par *Roboflow* sont en `640×640` pixels.

Le code ci-dessous effectue le téléchargement du *dataset* depuis GitHub, s'il n'est pas encore disponible localement, et réutilise `contar_shapes` pour vérifier les dimensions des images dans chaque *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"))
```

10.5.4.4 Comparaison d'une image de chaque *dataset*

Les deux *datasets* possèdent des images avec des résolutions différentes : `608×608` sur GitHub et `640×640` sur *Roboflow*. Pour une comparaison visuelle directe, les images sont redimensionnées à la même dimension avant d'être affichées côte à côte (Figure 10.38).

```
import cv2

if not chave.exists():
    print("Clé Roboflow introuvable : clé_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))

    # Redimensionne les deux à la même taille (la plus petite des deux)
    largura = min(img1.shape[1], img2.shape[1])
    altura = min(img1.shape[0], img2.shape[0])
    img1_r = cv2.resize(img1, (largura, altura))
    img2_r = cv2.resize(img2, (largura, altura))

    mm.show(
        [img1_r, img2_r],
        title=[f"Roboflow {img1.shape}", f"GitHub {img2.shape}"],
    )
```

Figure 10.38

10.5.4.5 Inférence avec le modèle entraîné

Le modèle entraîné dans la version 6 est utilisé pour effectuer l'inférence sur une image de test locale. La prédiction prend en compte les seuils de confiance et de chevauchement employés par la suppression des non-maxima (*Non-Maximum Suppression*, NMS), et le résultat est présenté sur l'image annotée de la Figure 10.39.

10.5.4.6 Évaluation des prédictions avec IoU et classe

Roboflow renvoie chaque boîte avec les coordonnées du centre (`x`, `y`) en *pixels* et la classe prédite, tandis que les étiquettes locales (`dados/dataset/test/labels/00001.txt`) suivent le format YOLO, avec le centre et les dimensions normalisés dans l'intervalle `[0, 1]`. Avant la comparaison à l'aide de `mm.IoU`, les boîtes doivent être converties dans le même format, avec les coordonnées du coin supérieur gauche et les dimensions exprimées en *pixels*.

Une prédiction n'est considérée comme correcte que lorsque la classe prédite coïncide avec la classe de la boîte réelle et que son *Intersection over Union* (IoU) est supérieure ou égale au seuil défini, en adoptant, dans cet exemple, `0,5` (50 %) comme valeur par défaut.

```python
# version.model est déprécié ; utiliser version.models()

if not chave.exists():
    print("Clé Roboflow introuvable : 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()  # inclut les boîtes détectées dans resp["predictions"]

    altura, largura, _ = mm.read(img_path).shape
```

```python
    print("Dimensions de l'image de test :", (altura, largura))

    pred.save("resultado.jpg")  # image annotée

    mm.show(
        mm.read("resultado.jpg"),
        title="Résultat de l'inférence avec le modèle Roboflow",
        figsize=(6, 6)
    )
```

Figure 10.39

! Le `class_id` de *Roboflow* ne correspond pas à l'index des *labels* locales

Lors de l'exportation, *Roboflow* réordonne les classes par **ordre alphabétique** dans le `data.yaml`, indépendamment de l'ordre utilisé dans le projet d'origine, conservé dans le `data.yaml` du GitHub. Ainsi, `class_id = 0` correspond à `Circulo` dans la réponse de l'API, tandis que le même index correspond à `Triangulo` dans les *labels* locales.
Par conséquent, la comparaison doit être effectuée par le **nom de la classe** (`p["class"]`), en le convertissant ensuite en index correspondant dans la liste locale. Les `class_id` ne doivent pas être comparés directement.

```python
# resp, largura e altura foram definidos na célula anterior

# Ordem das classes usada nas labels locais (dados/dataset/*/labels/*.txt)
CLASSES_LOCAIS = ["Triangulo", "Quadrado", "Pentagono", "Hexagono",
                  "Heptagono", "Circulo", "Elipse"]

def predicao_correta(pred: tuple, real: tuple, limiar: float = 0.5) -> bool:
    """True se mesma classe e mm.IoU(caixa_pred, caixa_real) >= limiar."""
    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:
    """Converte predição do Roboflow para (classe, (x, y, w, h))."""
    caixa = (p["x"] - p["width"] / 2, p["y"] - p["height"] / 2,
             p["width"], p["height"])
    # usa o nome da classe (não o class_id!) para casar com a ordem local
    classe = CLASSES_LOCAIS.index(p["class"])
    return (classe, caixa)

def carrega_labels_yolo(caminho_txt: str, largura: int, altura: int) -> list:
    """Lê rótulos YOLO (normalizados) e converte para (classe, (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("Chave do Roboflow não 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)} predições corretas (classe + IoU >= 50%)")
```

Pour visualiser le résultat, chaque prédiction est dessinée sur l'image : en vert lorsqu'il s'agit d'une réussite (classe + IoU seuil) et en rouge lorsqu'il s'agit d'une erreur, comme le montre la Figure 10.40..

```
import cv2

VERDE, VERMELHO = (0, 255, 0), (255, 0, 0)

if not chave.exists():
    print("Clé Roboflow introuvable : chave_roboflow.txt")
else:
    img_acertos = mm.read(img_path).copy()

    if not chave.exists():
        print("Clé Roboflow introuvable : 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)} prédictions correctes "
            f"(classe + IoU >= 50%)",
        figsize=(6, 6)
    )
```

Figure 10.40

10.5.5 Applications géométriques et intégrées

Le chapitre se termine en intégrant deux piliers de la VC : la **géométrie projective** (étudiée dans les chapitres 6 et 8) et l'**apprentissage profond**. La combinaison de ces approches soutient des applications pratiques dans le monde réel, comme illustré ci-dessous.

10.5.5.1 Réalité Augmentée avec Marqueurs et Homographie

Imaginez une caméra pointée vers une table sur laquelle quelqu'un a collé un petit marqueur ArUco. Selon l'angle de la caméra, ce marqueur apparaît **pivoté, incliné, en perspective** — jamais parfaitement carré. C'est précisément cette distorsion que l'homographie sait « lire » et corriger (ou, dans notre cas, répliquer vers une nouvelle image).

Le flux complet d'une application de RA basée sur des marqueurs suit trois étapes :

1. **Mise en situation :** le marqueur est inséré dans une scène réelle, subissant une transformation de perspective (simulant l'angle de la caméra).
2. **Détection :** l'algorithme localise le marqueur dans la scène et récupère les coordonnées exactes de ses 4 coins avec `cv2.aruco.ArucoDetector`.
3. **Substitution :** grâce à l'homographie entre le marqueur « idéal » et le marqueur « détecté », on projette une **nouvelle image virtuelle** exactement sur la zone du marqueur — comme si celui-ci s'était transformé en une fenêtre vers un autre contenu, comme le montre la Figure 10.41.

> **Note**
>
> **Détail technique important :** l'ArUco nécessite une marge blanche autour du motif (la « zone de silence ») pour que le détecteur puisse différencier le marqueur du fond. C'est pourquoi le code ci-dessous ajoute une bordure avec `cv2.copyMakeBorder` et utilise une interpolation `cv2.INTER_NEAREST` lors de la déformation de l'image, évitant ainsi que la rotation ne floute les petits carrés noirs et blancs et n'empêche la détection.

```
# 1. Génération du marqueur ArUco synthétique, déjà avec une marge blanche (zone calme)
# → cette marge est essentielle pour que le détecteur puisse « voir » le marqueur
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, avec 40px de cadre blanc de chaque côté
aruco = cv2.cvtColor(aruco_gray, cv2.COLOR_GRAY2BGR)
lado = aruco.shape[0]  # 280

# 2. Scène réelle de fond, en utilisant une image d'exemple de skimage.data
fundo = cv2.cvtColor(skdata.coffee(), cv2.COLOR_RGB2BGR)
cena = cv2.resize(fundo, (640, 480))

# Coins du marqueur « de face » (src) et sa position tournée/inclinée dans la scène (dst)
src = np.float32([[0, 0], [lado, 0], [lado, lado], [0, lado]])
dst = np.float32([[190, 160], [420, 70], [470, 330], [150, 370]])  # rotation + perspective

# 3. Projette le marqueur (avec des bords nets) sur la scène réelle
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]
```

```python
# 4. Détection du marqueur dans la scène (comme le ferait une caméra)
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. Image virtuelle (une autre image de skimage.data) qui va « remplacer » le marqueur
# Remarque : nous utilisons le carré INTÉRIEUR du marqueur (sans la marge) comme zone de
projection,
# donc l'homographie de l'image virtuelle utilise 'src' original (200x200), pas 'side' avec
bordure
src_interno = np.float32([[0, 0], [200, 0], [200, 200], [0, 200]])
virtual = cv2.resize(
    cv2.cvtColor(skdata.camera(), cv2.COLOR_RGB2BGR), (200, 200)
)

# Les coins détectés correspondent au marqueur AVEC la marge (280x280),
# donc nous recalculons H_ra en utilisant 'src' avec marge, pour garder la projection
cohérente
H_ra, _ = cv2.findHomography(src, corners[0][0])

# L'homographie détectée est appliquée à l'image virtuelle (redimensionnée à 'side')
virtual_grande = cv2.resize(virtual, (lado, lado))
ra = cena.copy()
mask_ra = cv2.warpPerspective(
    np.full((lado, lado), 255, np.uint8), H_ra, (640, 480), flags=cv2.INTER_NEAREST
)
ra[mask_ra > 0] = cv2.warpPerspective(
    virtual_grande, H_ra, (640, 480), flags=cv2.INTER_NEAREST
)[mask_ra > 0]

# Affichage : scène avec marqueur vs. scène avec Réalité Augmentée appliquée
mm.show(
    [cv2.cvtColor(cena, cv2.COLOR_BGR2RGB), cv2.cvtColor(ra, cv2.COLOR_BGR2RGB)],
    titles=["Marqueur dans la Scène Réelle (tourné)", "Superposition Virtuelle via
Homographie"],
    cols=2,
    figsize=(9, 4),
)
```

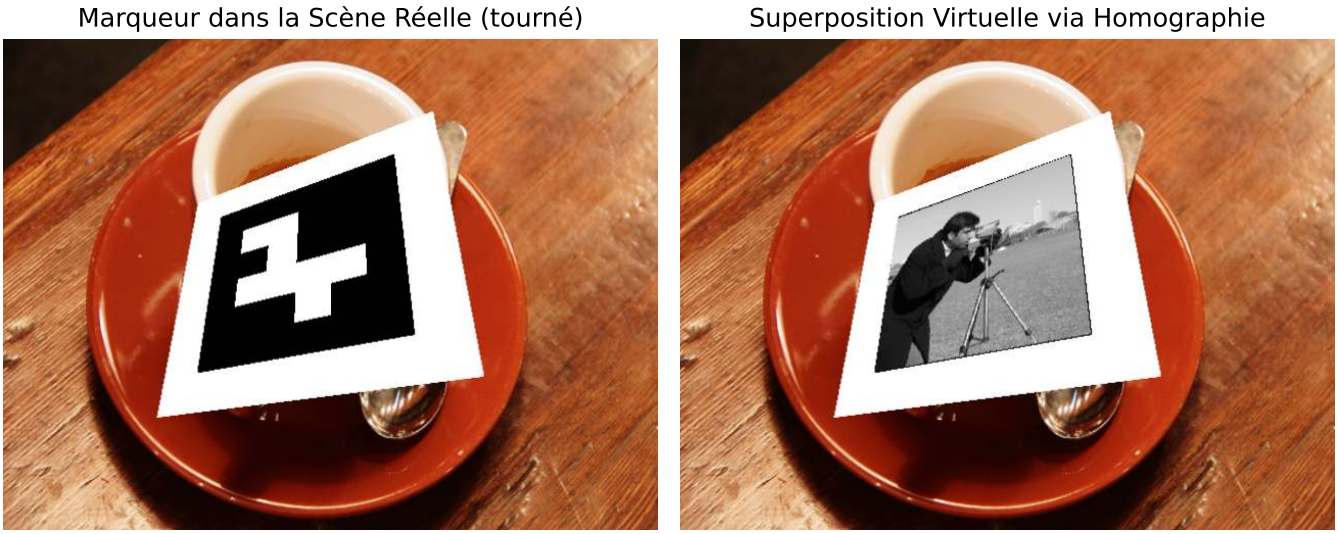


Figure 10.41: Réalité augmentée basée sur un marqueur ArUco : marqueur inséré dans une scène réelle et tourné, détecté et remplacé par une image virtuelle via homographie.

Résumé du *pipeline* :

Étape	Fonction	Clé
1. Génération	Marqueur ArUco avec marge blanche	`generateImageMarker` + `copyMakeBorder`
2. Mise en scène	Insère un marqueur déformé dans la scène	`findHomography` + `warpPerspective`
3. Détection	Localise le marqueur et retourne les coins	`ArucoDetector.detectMarkers`
4. Remplacement	Projette une image virtuelle sur le marqueur	`findHomography` + `warpPerspective`

💡 **Leçon :** le détecteur qui « ne trouve rien » est courant en vision par ordinateur. Demandez-vous toujours : **« ai-je fourni suffisamment de contraste et d'espace ? »** — cela vaut pour ArUco, les codes QR et la reconnaissance faciale.

10.5.5.2 Photogrammétrie et référence d'échelle

La **photogrammétrie** permet d'estimer les dimensions physiques d'objets à partir d'images numériques. Pour cela, on utilise un objet de référence aux dimensions connues, placé dans la même scène que l'objet d'intérêt. Cette procédure établit une relation entre les distances mesurées en *pixels* et leurs dimensions correspondantes dans le monde réel.

Considérons, par exemple, une carte de crédit, dont les dimensions suivent la norme internationale ISO/IEC 7810. Comme sa largeur est exactement de **8,56 cm**, il suffit de déterminer combien de *pixels* cette largeur occupe dans l'image pour calculer le facteur de conversion entre *pixels* et centimètres. Si la carte correspond à 140 *pixels*, alors chaque *pixel* représentera approximativement **0,061 cm**. Ce même facteur d'échelle peut être appliqué pour estimer les dimensions de tout autre objet situé dans le même plan de la scène, comme illustré dans la Figure 10.42.

La procédure peut être divisée en deux étapes principales :

1. **Segmentation et *bounding box* :** localiser, dans l'image, à la fois l'objet de référence et l'objet d'intérêt, en utilisant des techniques telles que la segmentation par couleur, le seuillage, la détection de contours ou des méthodes de détection d'objets.

2. **Conversion en dimensions physiques :** calculer le rapport cm/pixel à partir de la largeur connue de l'objet de référence et l'utiliser pour convertir les mesures de l'objet d'intérêt de *pixels* en centimètres.

> **Note**
>
> **Condition pour des mesures fiables**
> La conversion entre *pixels* et centimètres suppose que l'objet de référence et l'objet d'intérêt soient approximativement dans le même plan et à la même distance de la caméra. Dans ces conditions, l'échelle reste pratiquement constante sur toute l'image. Les différences de profondeur, l'inclinaison de la caméra ou les distorsions de l'objectif peuvent introduire des erreurs dans les mesures estimées.

```python
COR_REFERENCIA = (200, 200, 200)  # Carte de référence (grise)
COR_OBJETO = (60, 60, 220)  # Objet cible (rouge)

# Dessin de la scène synthétique
cena_medicao = np.full((300, 500, 3), 255, dtype=np.uint8)
cv2.rectangle(
    cena_medicao, (30, 200), (30 + 140, 200 + 88), COR_REFERENCIA, -1
)
cv2.rectangle(cena_medicao, (250, 100), (250 + 220, 100 + 150), COR_OBJETO, -1)


def caixa_delimitadora_por_cor(imagem_bgr, cor_bgr, tolerancia=40):
    diferenca = np.abs(imagem_bgr.astype(int) - np.array(cor_bgr)).sum(axis=2)
    mascara = (diferenca < tolerancia).astype(np.uint8) * 255
    contornos, _ = cv2.findContours(
        mascara, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
    )
    maior_contorno = max(contornos, key=cv2.contourArea)
    return cv2.boundingRect(maior_contorno)


x_ref, y_ref, w_ref_px, h_ref_px = caixa_delimitadora_por_cor(
    cena_medicao, COR_REFERENCIA
)
x_obj, y_obj, w_obj_px, h_obj_px = caixa_delimitadora_por_cor(
    cena_medicao, COR_OBJETO
)

# Calcul d'échelle physique
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

# Dessin des boîtes et mesures estimées
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=[
        "Scène originale",
        "Mesure par référence d'échelle (centimètres)",
    ],
    cols=2,
    figsize=(9, 4),
)
```

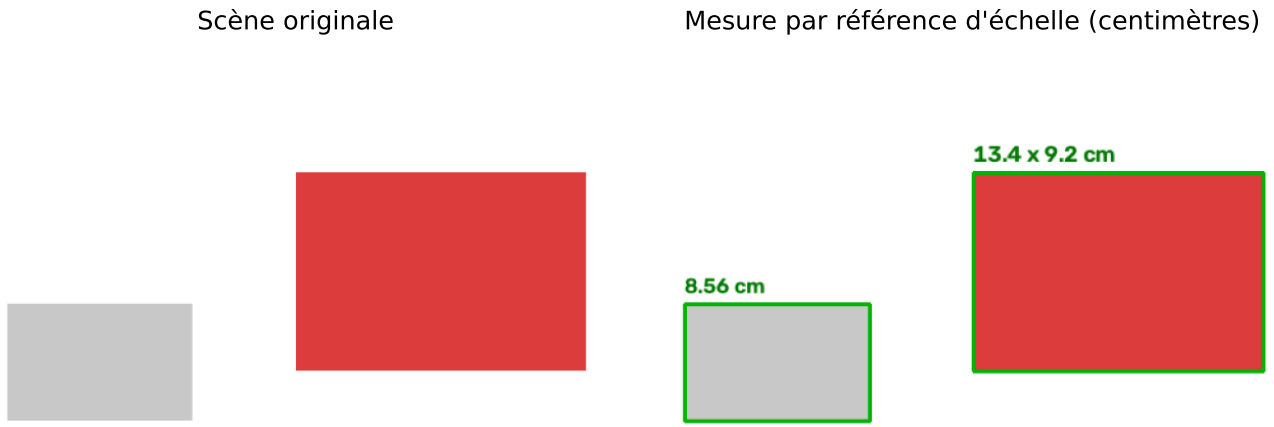


Figure 10.42: Mesure de dimensions physiques réelles à l'aide d'une carte de référence d'échelle connue (8,56 cm) rendue. Le rectangle gris représente la carte de référence et le rectangle bleu représente l'objet cible. Les dimensions estimées de l'objet sont affichées en centimètres.

Résumé du *pipeline* :

Étape	Ce qu'elle fait	Fonction clé
1. Scène synthétique	Dessine la carte de référence et l'objet cible avec des couleurs distinctes	`cv2.rectangle`
2. Segmentation	Isole chaque objet par couleur et extrait son contour	`cv2.findContours`

Étape	Ce qu'elle fait	Fonction clé
3. Boîte englobante	Obtient la boîte englobante (position et taille en pixels) de chaque objet	`cv2.boundingRect`
4. Mise à l'échelle	Convertit les pixels en centimètres en utilisant la largeur connue de la carte	Règle de trois : $\text{cm/pixel} = \dfrac{8{,}56}{w_{ref_px}}$
5. Annotation	Dessine les boîtes et affiche les mesures estimées sur l'image	`cv2.rectangle` + `cv2.putText`

💡 **Application dans le monde réel :** c'est exactement la technique utilisée par les *apps* de *e-commerce* qui estiment la taille d'un produit à partir d'une photo prise à côté d'une carte, par les systèmes agricoles qui mesurent des fruits sur des tapis roulants, et même par les expertises forensiques qui calculent les dimensions de traces sur des scènes de crime — tout repose sur la même idée : **une règle connue dans la photo elle-même.**

10.6 Résumé

Ce chapitre, qui clôt la deuxième partie de l'ouvrage, a présenté :

- **Convolution et *pooling* appris :** la même opération mathématique de convolution qu'au chapitre 3, mais avec des *kernels* traités comme des paramètres ajustés par l'entraînement, au lieu d'être définis manuellement ;
- **Le partage des poids et la hiérarchie des caractéristiques** comme propriétés qui rendent les CNN efficaces et capables d'apprendre des représentations de plus en plus abstraites dans des couches successives ;
- **L'entraînement d'une CNN de zéro**, avec des performances comparables — et pas nécessairement supérieures — aux classificateurs classiques du chapitre 7 sur une base petite et simple, soulignant que le choix de la méthode doit être proportionnel à la complexité réelle du problème ;
- **Le transfert d'apprentissage**, démontré expérimentalement comme une stratégie efficace pour des tâches avec peu de données étiquetées, réutilisant un extracteur de caractéristiques déjà entraîné sur une tâche connexe, et ses limites, mises en évidence par le transfert négatif entre des domaines très distincts ;
- **Des applications à grande échelle** avec des modèles pré-entraînés de classification, de détection (*Faster R-CNN*) et de segmentation (*DeepLabV3*), suivant le même principe de transfert d'apprentissage à l'échelle industrielle ;
- **Le transfert d'apprentissage appliqué à la détection d'objets**, en ajustant un YOLO pré-entraîné sur COCO pour localiser et classer des objets géométriques synthétiques, illustrant le même principe de gel partiel dans un domaine source et cible encore plus éloignés l'un de l'autre ;
- **La segmentation sémantique avec U-Net**, implémentée et entraînée de zéro sur un ensemble synthétique à faible contraste, surpassant une ligne de base classique de seuillage grâce aux connexions de raccourci entre l'encodeur et le décodeur ;
- **L'ingénierie des données pour la vision par ordinateur avec *Roboflow***, en utilisant un *dataset* et un modèle pré-entraîné pour effectuer une inférence sur la détection d'objets géométriques, ainsi qu'en comparant des ensembles de données avec des résolutions différentes, mais avec les mêmes classes d'objets ;
- L'intégration de la **géométrie computationnelle** (homographie et calibrage) et de l'**apprentissage profond** dans deux applications réelles qui clôturent l'ouvrage : la **réalité augmentée** et la **photogrammétrie**.

10.7 🤖 Utilisation de Gemini Notebook comme tuteur complémentaire

Dans cette édition, l'utilisation de **Gemini Notebook** est encouragée comme outil d'apprentissage complémentaire. Fondé sur l'intelligence artificielle, le système utilise exclusivement les documents fournis par l'auteur comme source de connaissance, produisant des réponses alignées sur le contenu et l'approche adoptés tout au long de ce chapitre.

🎓 Étudiez avec le tuteur intelligent

🚀 ACCÉDER À GEMINI NOTEBOOK : CHAPITRE 09

🌐 **Langue et langage de programmation**

Le projet de ce chapitre dans Gemini Notebook a été construit uniquement avec le texte en **portugais** et les exemples de code en **Python**. Si vous étudiez à partir de l'édition en anglais ou en français, ou si vous suivez le parcours en C++, les réponses du tuteur peuvent ne pas correspondre exactement à la version que vous lisez.

⚠️ **Avis concernant le contenu généré par l'IA**

Bien qu'il s'agisse d'un outil précieux de soutien à l'étude, Gemini Notebook peut éventuellement produire des réponses incomplètes, imprécises ou incorrectes. Il est recommandé de valider les informations en consultant le matériel du chapitre, les livres, les articles scientifiques et d'autres sources académiques fiables. Dans la mesure du possible, exécutez et expérimentez les exemples pratiques présentés tout au long du texte afin de consolider la compréhension des concepts.

10.8 Liste d'exercices

Les exercices suivants consolident les concepts présentés dans ce chapitre à travers des adaptations, des expériences et des extensions des algorithmes développés tout au long du texte, en utilisant les bibliothèques `PyTorch`, `ultralytics` et la bibliothèque didactique `morph`.

1. **(10%)** Étudiez l'impact de la profondeur dans une architecture convolutionnelle. En partant du réseau à deux couches du Projet Pratique 1, ajoutez une troisième couche convolutionnelle avec 32 filtres avant les couches entièrement connectées. Entraînez la nouvelle architecture en conservant le même nombre d'époques et la même répartition des données. Comparez la précision sur l'ensemble de test et le nombre total de paramètres entraînables par rapport au réseau original, en discutant si l'augmentation de la profondeur a apporté un bénéfice mesurable pour des images de dimension 8×8.

2. **(15%)** Évaluez le seuil de données nécessaires dans le domaine cible pour que l'entraînement d'un CNN de zéro devienne compétitif avec le **transfert d'apprentissage**. En faisant varier le nombre d'échantillons d'entraînement disponibles dans le domaine B pour $\{5, 10, 20, 40, 80\}$, mesurez la précision de test pour les deux stratégies. Présentez les résultats dans un graphique en lignes et déterminez à partir de quel volume de données l'entraînement de zéro atteint des performances équivalentes à celles de l'extracteur pré-entraîné.

3. **(15%)** Étudiez la stratégie de **réglage fin partiel** (*fine-tuning*) par rapport au gel total des poids. Dans le scénario de transfert d'apprentissage entre domaines de chiffres, décongelez la deuxième couche convolutionnelle (`conv2`) de l'extracteur afin qu'elle soit mise à jour conjointement avec la tête de classification pendant l'entraînement dans le domaine B. Comparez la précision obtenue avec le gel total et avec l'entraînement de zéro, en discutant du compromis entre la capacité d'adaptation et le risque de sur-ajustement (*overfitting*).

4. **(20%)** Évaluez l'influence de la profondeur du gel (*freeze*) sur les performances des détecteurs **YOLO** soumis au transfert d'apprentissage. En utilisant l'ensemble de données synthétique de formes géométriques, exécutez le réglage fin en faisant varier le paramètre de gel du *backbone* pour $\{0, 5, 10, 15\}$.

Enregistrez la métrique mAP_{50} sur l'ensemble de validation pour chaque configuration, présentez les données dans un tableau et discutez si le gel partiel est avantageux lorsque les domaines source (COCO) et cible (formes géométriques) sont significativement distincts.

5. **(20%)** Étudiez l'importance des **connexions de raccourci** (*skip connections*) dans l'architecture **U-Net** pour la segmentation sémantique. Implémentez une variante `UNetSemAtalhos` qui fonctionne comme un *autoencodeur* convolutionnel traditionnel, en supprimant les concaténations entre les étages de l'encodeur et du décodeur. Entraînez les deux modèles sur la même base de nodules synthétiques, comparez l'IoU moyen sur l'ensemble de validation et présentez visuellement la différence dans la précision des contours segmentés par chaque méthode.

6. **(20%) — Défi : segmentation sémantique avec *Roboflow*.** Utilisez un projet *Roboflow* de type *instance segmentation*, contenant les sept classes de formes géométriques utilisées dans ce chapitre. Exportez l'*ensemble de données* au format `coco-segmentation` et développez une procédure pour convertir les polygones stockés dans les fichiers `_annotations.coco.json` en masques sémantiques multiclasses, dans lesquels chaque pixel reçoit l'indice de la classe correspondante et la valeur `0` représente le fond. Utilisez les images et les masques résultants pour entraîner la `UNetCompacta`. Évaluez l'IoU moyen sur l'ensemble de test et comparez visuellement les masques prédits avec les annotations originales. Discutez des principales difficultés rencontrées lors de la conversion des annotations COCO en masques et des effets des objets superposés ou appartenant à différentes classes.

7. **(Bonus – 10%)** Développez un système interactif qui combine la **détection d'objets (YOLO)** avec la **mesure par référence d'échelle (photogrammétrie)**. Entraînez le détecteur pour identifier deux classes dans une scène : une « Carte de Référence » (dimension connue de 8,56 cm × 5,39 cm) et un « Objet Cible ». Lors de l'inférence sur une nouvelle image, utilisez la dimension en pixels de la *boîte englobante* de la carte détectée pour convertir les dimensions de la boîte de l'objet cible en centimètres. Affichez l'image traitée avec les étiquettes de classe, la probabilité de confiance et les dimensions physiques estimées superposées.

8. **(Bonus – 10%)** Développez un système d'**estimation de profondeur par vision stéréo** à partir de deux images de la même scène obtenues depuis des positions différentes, simulant une paire de caméras stéréo. Considérez que la distance entre les deux positions de capture (*baseline*) est connue.

   Utilisez l'une des méthodes de **détection d'objets** présentées dans le chapitre, comme YOLO, pour localiser les objets d'intérêt dans les deux images. Pour chaque détection, établissez la correspondance entre le même objet dans les deux positions et déterminez sa **disparité**. À partir de la disparité, de la *baseline* et des paramètres de la caméra, utilisez la géométrie stéréo pour estimer la distance de chaque objet par rapport aux caméras.

   Comme extension de la bibliothèque didactique `morph`, modifiez la méthode `showBoundBox` afin qu'en plus de la **classe** et de la **confiance de la détection**, elle présente sur chaque *boîte englobante* la **distance estimée de l'objet**. Le résultat doit permettre de visualiser directement, sur les images, la classe, la précision (confiance) et la profondeur de chaque objet détecté.

   Présentez les deux images avec les détections, les correspondances entre les objets, l'image de disparité et une représentation de la profondeur estimée. Discutez de la manière dont la distance entre les caméras, la précision de la détection et de la correspondance, la résolution des images et la position de l'objet dans la scène influencent la qualité de l'estimation.

   Pour la validation, utilisez au moins un objet dont la distance à la caméra est connue. Comparez la profondeur estimée avec la valeur réelle et rapportez l'**erreur absolue** et l'**erreur relative**. Discutez également des limites de la méthode lorsqu'un objet n'est pas correctement détecté dans les deux images ou lorsque la correspondance entre les régions observées est ambiguë.

10.9 Clôture de la Partie II

Ce chapitre conclut la Partie II du livre et clôt la séquence de contenus initiée au chapitre 6, consacrée à la représentation, la détection, la description et la mise en correspondance de caractéristiques dans les images. Tout au long de ces chapitres, des méthodes classiques de vision par ordinateur (VC) basées sur des **caractéristiques conçues manuellement**, telles que Sobel, LBP, HOG, ORB et Haar Cascade, ainsi que des méthodes fondées sur des **caractéristiques apprises automatiquement**, représentées par les CNN, ont été présentées.

Les exemples et expériences développés montrent qu'aucune de ces approches n'est universellement supérieure. Le choix de la technique la plus adaptée dépend des caractéristiques du problème, de la disponibilité des données d'entraînement, des exigences de précision et des contraintes computationnelles de l'application. Dans des problèmes bien structurés et avec peu de données, les descripteurs classiques offrent souvent des solutions simples et efficaces. En revanche, des tâches plus complexes tendent à bénéficier de la capacité d'apprentissage offerte par les CNN.

Plusieurs directions d'étude peuvent approfondir les concepts présentés dans cette partie du livre, parmi lesquelles :

- **Les architectures modernes de CNN**, telles que ResNet, EfficientNet et les *Vision Transformers*, qui élargissent la capacité de représentation et les performances dans les tâches de classification et de reconnaissance visuelle ;
- **La détection et la segmentation d'objets**, avec un accent sur les familles YOLO et les modèles de segmentation basés sur des *prompts*, comme *Segment Anything* ;
- **La reconstruction tridimensionnelle et le SLAM** (*Simultaneous Localization and Mapping*), qui utilisent plusieurs images pour estimer la géométrie de la scène et la trajectoire de caméras en mouvement ;
- **Les modèles génératifs d'images**, tels que les réseaux adversariaux génératifs (GAN) et les modèles de diffusion, capables de synthétiser des images réalistes à partir d'exemples ou de descriptions textuelles.

Les fondements développés tout au long de la Partie II constituent la base de ces domaines et d'autres domaines avancés de la VC, dans lesquels la représentation adéquate des informations visuelles demeure un élément central pour l'analyse et la compréhension des images.

Références du chapitre

Les concepts et algorithmes présentés dans ce chapitre s'appuient sur des références classiques et contemporaines de la littérature sur l'apprentissage profond appliqué à la vision par ordinateur :

- Mcculloch (1943), pour la proposition de la première abstraction mathématique et logique du neurone artificiel, posant les fondements conceptuels du traitement neuronal computationnel.
- Rosenblatt (1958), pour la formulation originale du **Perceptron**, modèle précurseur du neurone artificiel utilisé dans les architectures modernes d'apprentissage profond.
- Goodfellow (2016) et Lecun (2015), pour les fondements des réseaux de neurones, de la convolution, des fonctions d'activation et de l'entraînement des modèles profonds.
- Bishop (2006), pour les concepts rigoureux de reconnaissance des formes, de probabilité, d'estimation du maximum de vraisemblance et de méthodes statistiques appliquées à l'apprentissage automatique.
- Ronneberger (2015), pour l'architecture **U-Net**, utilisée dans la segmentation sémantique avec des connexions de raccourci entre l'encodeur et le décodeur.
- Redmon (2016), pour l'architecture **YOLO** (*You Only Look Once*), utilisée dans les expériences de détection d'objets avec transfert d'apprentissage.
- Ren (2015), pour l'architecture **Faster R-CNN**, employée comme modèle pré-entraîné de détection d'objets.
- He (2016), pour l'architecture **ResNet**, base de divers extracteurs de caractéristiques pré-entraînés utilisés dans ce chapitre.

- Chen (2018), pour l'architecture **DeepLabV3**, utilisée comme modèle pré-entraîné de segmentation sémantique.
- Kirillov (2023), pour le modèle **Segment Anything (SAM)**, mentionné comme piste d'étude pour la segmentation promptable.
- {google} (2025), concernant l'outil **Gemini Notebook**, utilisé dans l'élaboration de l'infographie de synthèse du chapitre et mis à disposition comme support complémentaire à l'étude.

Exécuter Colab | Ouvrir GitHub

10.10 💻 Partie Pratique avec des Exercices de Programmation

La présente liste d'**Exercices de Programmation (EP)** consolide les formulations théoriques présentées tout au long du Chapitre 9 — Apprentissage Profond pour la Vision par Ordinateur — à travers un parcours pratique appliqué. Contrairement à l'entraînement de réseaux de neurones complets avec PyTorch, qui exige du temps d'exécution et parfois un GPU, les EP de ce chapitre isolent les **grandeurs intermédiaires** d'un *pipeline* réel d'apprentissage profond — la sortie d'une seule couche convolutive, le résultat d'une opération de *pooling*, le comptage des paramètres entraînables d'une architecture, le chevauchement entre boîtes englobantes candidates, la qualité d'un masque de segmentation et le filtre de suppression des non-maximums — permettant de valider manuellement chaque étape du raisonnement sans dépendre de bibliothèques d'apprentissage automatique ni d'un entraînement réel.

L'enchaînement des exercices reproduit le flux conceptuel du chapitre et croît en difficulté à chaque étape : on commence par le calcul manuel de la sortie d'une **couche convolutive apprise** (🟢), à partir d'un *kernel* et d'un biais déjà entraînés ; on avance vers l'opération de ***pooling*** (🟢, maximum et moyenne), qui réduit la résolution spatiale entre les blocs convolutifs ; on poursuit avec le **comptage des paramètres entraînables** (🟡) d'une architecture CNN complète, mettant en évidence pourquoi le partage des poids rend ces réseaux tellement plus économes qu'une couche entièrement connectée équivalente ; on approfondit le calcul de l'**Intersection sur Union (IoU)** et de la **Suppression des Non-Maximums (NMS)** (🟡), étape de post-traitement commune aux détecteurs comme Faster R-CNN et YOLO ; on passe à l'**évaluation des masques de segmentation** (🟠) avec les mêmes métriques IoU et Dice utilisées pour comparer U-Net à la ligne de base morphologique classique ; et on conclut avec un ***pipeline* intégré** (🔴), reliant la sortie d'un détecteur d'objets (après NMS) à une mesure du monde réel par référence d'échelle — le même principe que la photogrammétrie étudiée dans l'intégration finale du chapitre.

Chaque fois que cela a du sens, chaque exercice indique des méthodes de la bibliothèque didactique `morph.py` (la même utilisée tout au long du chapitre, importée sous le nom `mm`) qui résolvent une étape du problème ou servent de référence pour vérifier vos propres calculs — sans toutefois remplacer le raisonnement que vous devez implémenter.

❗ Directives pour la Résolution des Exercices de Programmation

Dans tous les exercices de ce chapitre, les étapes de discrétisation ou d'arrondi numérique doivent employer l'arrondi standard à l'entier le plus proche (*round half away from zero*), atténuant les ambiguïtés pour les valeurs dont la fraction est exactement égale à 0,5. Sauf indication explicite contraire : (i) l'opération de « convolution » suit la convention adoptée par les *frameworks* d'apprentissage profond — **corrélation croisée**, sans inversion spatiale du *kernel*, exactement comme présenté dans la section « Couche convolutive » ; (ii) le remplissage (*padding*) se fait avec des zéros ; (iii) les boîtes englobantes sont spécifiées au format coin-à-coin (x_1, y_1, x_2, y_2), avec $x_1 < x_2$ et $y_1 < y_2$; et (iv) les vecteurs/matrices suivent une indexation à partir de 0, avec la convention `[ligne][colonne]` pour les structures bidimensionnelles.

🎯 Objectif de ce Carnet

Le carnet permet de développer, valider, organiser et tester des solutions d'**Exercices de Programmation (EPs)** dans des environnements interactifs, comme Colab, avec les mêmes cas de test que Moodle, en les y copiant uniquement au moment d'enregistrer la note officielle.

Téléchargement

Téléchargez `morph.py` et `testsuite.py` en exécutant la cellule ci-dessous :

```
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
```

```
Environnement prêt. Morph : 1.1.9 | OpenCV : 5.0.0 | TestSuite : 1.1.2
```

Exécution des tests

Pour évaluer les tests, exécutez `TestSuite("EP09_01.extensão").run()` dans une nouvelle cellule, en remplaçant l'extension par celle du langage utilisé (`.py`, `.java`, `.c`, `.cpp`, `.js` ou `.r`). Le système télécharge les cas de test depuis GitHub, exécute le programme et calcule automatiquement la note.

Pour tester directement du code Python, sans enregistrer de fichier, utilisez `run_code(codigo)` en passant le code comme *chaîne de caractères* dans une variable `codigo` :

```
codigo = """
# ... votre code ici ...
"""
TestSuite("EP09_01").run_code(codigo)
```

10.10.1 EP09_01 🟢 Convolution 2D Manuelle (*Forward* d'une Couche Apprise)

PyTorch, présenté dans ce chapitre, exécute `nn.Conv2d(x)` en un seul appel — mais derrière cela se cache simplement la corrélation croisée entre un *kernel* (déjà entraîné) et un voisinage de l'entrée, suivie de la somme d'un biais et d'une activation, exactement comme formalisé dans la Section « Couche Convolutionnelle ». La différence essentielle par rapport à la convolution de *kernels* fixes du Chapitre 3 est qu'ici, les valeurs du *kernel* et du biais **sont déjà prêtes** (comme si elles avaient été apprises par descente de gradient), et il vous revient de reproduire manuellement la passe directe (*forward pass*) que le *framework* exécute en interne.

Avant d'entraîner une véritable CNN, vous êtes chargé d'implémenter cette passe directe à partir de zéro, pour une seule couche convolutionnelle avec un seul canal d'entrée et un seul filtre de sortie, incluant la prise en charge d'un *padding* et d'un *stride* arbitraires.

10.10.1.1 📋 Directives d'Implémentation

1. **Entrée :** Lire les dimensions $H \times W$ de la carte de caractéristiques d'entrée, puis ses $H \times W$ valeurs réelles.

2. ***Kernel* et biais :** Lire les dimensions $k_h \times k_w$ du *kernel* (déjà entraîné), ses valeurs réelles, et le biais b (réel, scalaire).

3. **Hyperparamètres :** Lire le *padding* p (entier, nombre de zéros ajoutés à chaque bord) et le *stride* s (entier, pas de glissement).
4. **Remplissage :** Ajouter p zéros sur chacune des quatre bordures de la carte d'entrée avant la corrélation.
5. **Corrélation croisée :** Pour chaque position de sortie (i, j), calculer

$$z(i,j) = b + \sum_{u=0}^{k_h-1} \sum_{v=0}^{k_w-1} K(u,v) \cdot X_{pad}(i \cdot s + u,\ j \cdot s + v),$$

   en parcourant l'entrée **sans** inverser le *kernel* (convention des *frameworks* d'apprentissage profond, différente de la convolution mathématique classique).
6. **Activation :** Appliquer ReLU à chaque valeur : $a(i,j) = \max(0, z(i,j))$.
7. **Dimensions de sortie :** $O_h = \lfloor (H + 2p - k_h)/s \rfloor + 1$ et $O_w = \lfloor (W + 2p - k_w)/s \rfloor + 1$.
8. **Sortie :** Imprimer O_h et O_w sur la première ligne, suivis de O_h lignes avec chacune O_w valeurs réelles (la carte de caractéristiques de sortie, déjà avec ReLU appliquée), formatées avec 4 décimales.

10.10.1.2 📌 Contraintes Computationnelles

- **Un canal d'entrée, un filtre de sortie :** Il n'est pas nécessaire de gérer plusieurs canaux ou plusieurs filtres dans cette version simplifiée.
- **Sans inversion du *kernel* :** Implémentez la corrélation croisée, et non la convolution mathématique classique avec *kernel* inversé — c'est l'opération que PyTorch (et la plupart des *frameworks*) appelle « convolution ».
- **Remplissage par zéros :** Les p pixels ajoutés à chaque bord valent toujours 0.
- **Formatage :** Toutes les valeurs de sortie doivent avoir exactement 4 décimales, même lorsque la valeur est entière (ex. : `2.0000`).

10.10.1.3 🧠 Fondements Théoriques

Élément	Rôle dans la couche convolutionnelle
Kernel K	Paramètres appris par descente de gradient, analogues aux coefficients d'un filtre fixe du Chapitre 3, mais ajustés par rétropropagation
Biais b	Décalage appris, ajouté après la corrélation — permet au neurone de « s'activer » même avec une entrée nulle
Padding	Contrôle la dimension spatiale de sortie et évite la perte d'information aux bords à chaque couche
Stride	Contrôle le pas du déplacement ; des valeurs > 1 réduisent la résolution spatiale, comme une forme de sous-échantillonnage intégrée à la convolution elle-même
ReLU	Introduit une non-linéarité après la combinaison linéaire, exactement comme dans la Section « Fonction d'Activation »

10.10.1.4 🧩 Méthodes de `morph.py` pouvant aider

- `mm.readImg(h, w, dtype='float')` — lit directement une matrice $h \times w$ de valeurs réelles depuis l'entrée standard, évitant ainsi le *parsing* manuel de la carte de caractéristiques et du *kernel*.

- `mm.correlacao0(f, kernel, bias)` — implémente la même somme de corrélation croisée + biais que vous allez calculer à la main, mais **sans** prise en charge du *padding* ou du *stride*, et convertit le résultat en `uint8` (tronque les valeurs négatives et décimales). Peut servir de référence conceptuelle ou pour vérifier le cas le plus simple ($p = 0$, $s = 1$), mais ne remplace pas votre implémentation complète — qui doit préserver le signe, les décimales, le *padding*, le *stride* et ReLU.

10.10.1.5 📦 Spécification d'Entrée et de Sortie (VPL)

Entrée :

- Ligne 1 : Entiers H et W.
- Les H lignes suivantes : W valeurs réelles chacune (carte d'entrée).
- Ligne suivante : Entiers k_h et k_w.
- Les k_h lignes suivantes : k_w valeurs réelles chacune (*kernel*).
- Ligne suivante : Réel b (biais).
- Ligne suivante : Entiers p et s.

Sortie :

- Ligne 1 : Entiers O_h et O_w.
- Les O_h lignes suivantes : O_w valeurs réelles chacune, avec 4 décimales.

10.10.1.6 📌 Exemples

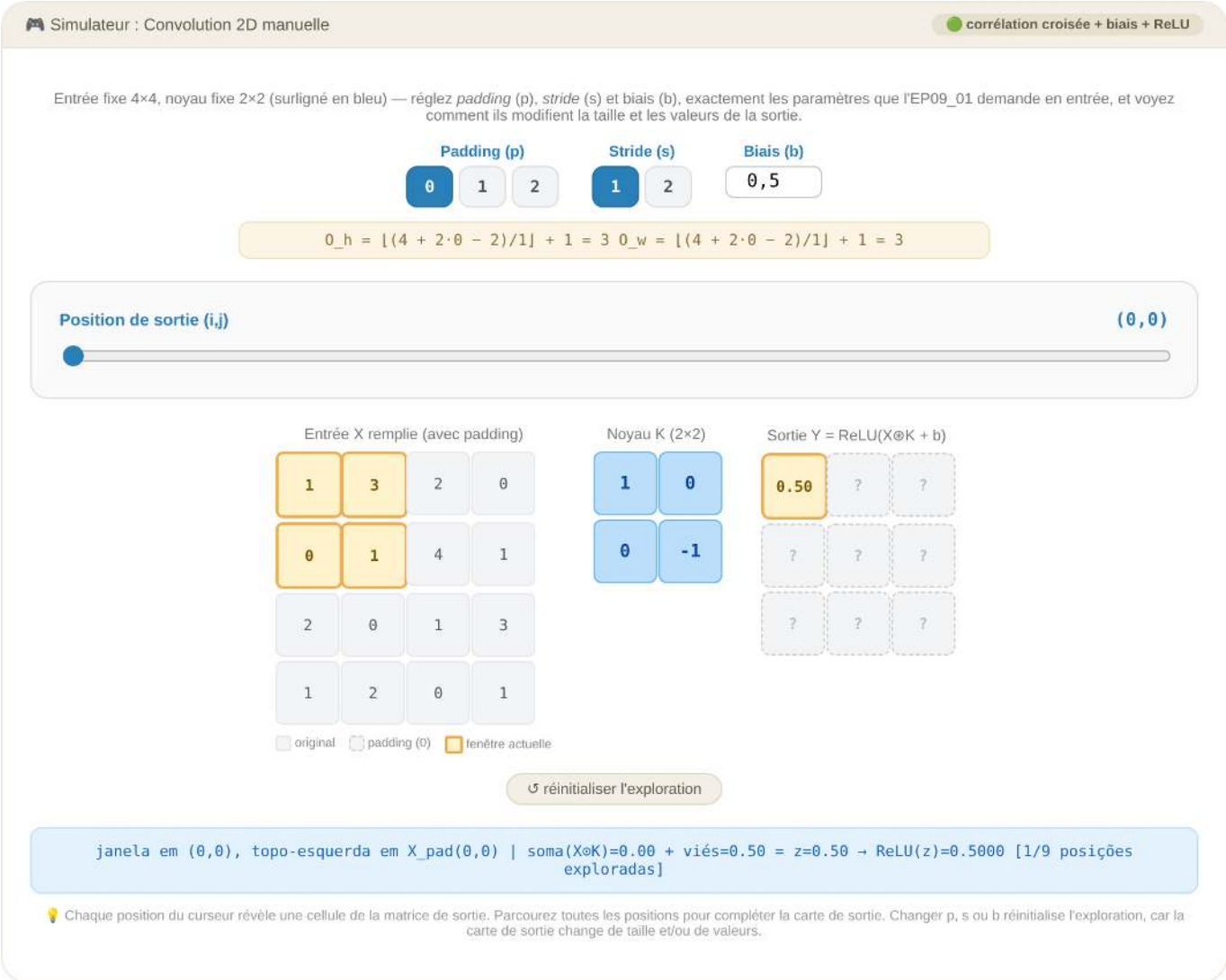
Entrée	Sortie	Observation
3 3 1 2 0 0 1 2 1 0 1 2 2 1 1 1 1 -2 0 1	2 2 2.0000 3.0000 0.0000 2.0000	*Padding* 0, *stride* 1 : sortie 2×2 sans remplissage.
3 3 1 2 0 0 1 2 1 0 1 2 2 1 0 0 1 0 1 2	2 2 1.0000 0.0000 1.0000 2.0000	*Padding* 1, *stride* 2 : entrée remplie de zéros avant la corrélation.

```
%%writefile EP09_01.py
# Code Python
```

```
Overwriting EP09_01.py
```

```
TestSuite("EP09_01.py").run()
```

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



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap09/sim-ep0901.html

Figure 10.43: Simulateur EP09_01 : Convolution 2D manuelle (corrélation croisée + biais + ReLU, avec *padding* et *stride* ajustables)

10.10.2 EP09_02 🟢 *Pooling* manuel (Maximum et moyenne)

Entre les blocs convolutifs, l'architecture typique d'un CNN intercale des couches de ***pooling***, qui réduisent la résolution spatiale de la carte de caractéristiques sans introduire de nouveaux paramètres entraînables — contrairement à la convolution, le *pooling* n'a pas de poids : il résume simplement chaque fenêtre de l'entrée à une seule valeur, par un maximum ou par une moyenne, exactement comme formalisé dans la section "*Pooling*".

Vous êtes chargé d'implémenter cette opération à partir d'une fenêtre glissante carrée, sans chevauchement partiel sur les bords (uniquement des fenêtres complètes), en supportant les deux types les plus courants : `max` (préserve la valeur la plus saillante, typiquement utilisée pour retenir les bords et les textures fortes) et `avg` (lisse la région, préservant l'information d'intensité moyenne).

10.10.2.1 📋 Directives d'implémentation

1. **Entrée :** Lire les dimensions $H \times W$ de la carte de caractéristiques d'entrée et ses $H \times W$ valeurs réelles.
2. **Fenêtre :** Lire les entiers k (taille de la fenêtre carrée $k \times k$) et s (*stride*).
3. **Type :** Lire une *chaîne* de caractères, `max` ou `avg`, indiquant le type de *pooling*.
4. **Sans remplissage :** Cette opération **n'utilise pas** de *padding* ; les fenêtres qui dépasseraient le bord de l'entrée sont ignorées.
5. **Calcul :** Pour chaque position de sortie (i, j), calculer le maximum ou la moyenne des $k \times k$ valeurs de la fenêtre correspondante, commençant à $(i \cdot s,\ j \cdot s)$.
6. **Dimensions de sortie :** $O_h = \lfloor (H-k)/s \rfloor + 1$ et $O_w = \lfloor (W-k)/s \rfloor + 1$.
7. **Sortie :** Imprimer O_h et O_w sur la première ligne, suivis de O_h lignes avec O_w valeurs réelles chacune, formatées avec 4 décimales.

10.10.2.2 📌 Contraintes computationnelles

- **Fenêtre carrée :** $k \times k$, sans support pour les fenêtres rectangulaires dans cette version.
- **Sans *padding* :** seules les fenêtres entièrement contenues dans l'entrée sont considérées — les dimensions "restantes" sont simplement ignorées.
- **`avg` utilise une division réelle :** la moyenne est toujours somme$/k^2$, même lorsque le résultat a de nombreuses décimales — arrondissez uniquement lors du formatage final, conformément à la directive générale du chapitre.
- **Formatage :** toutes les valeurs de sortie avec exactement 4 décimales.

10.10.2.3 🧠 Fondement théorique

Élément	Rôle dans l'architecture
Pooling maximum	Préserve l'activation la plus forte de la fenêtre ; courant après les couches convolutives pour retenir les bords et les textures saillantes
Pooling moyen	Lisse la région, préservant l'intensité moyenne ; courant dans les couches finales (*global average pooling*)
Absence de paramètres	Différencie le *pooling* de la convolution : réduit la résolution spatiale sans coût supplémentaire d'entraînement
Réduction de résolution	Contribue à l'invariance aux petites translations et à la réduction du coût computationnel des couches suivantes

10.10.2.4 🧩 Méthodes de `morph.py` qui peuvent aider

Le `morph.py` n'implémente pas le *pooling* avec sous-échantillonnage directement, mais deux familles d'opérations montrent la même idée sous un autre angle, utile pour vérifier votre intuition :

- `mm.dil(f, Bc)` / `mm.dil0(f, B)` — dilatation morphologique : remplace chaque pixel par le **maximum** de son voisinage défini par l'élément structurant B (ex. : `mm.sebox(n)` pour une fenêtre $(2n+1) \times (2n+1)$). Conceptuellement, c'est un "*max-pooling* sans sous-échantillonnage" (produit une image de même taille, au lieu d'une image réduite).
- `mm.blur(f, N)` — lissage par moyenne dans une fenêtre $N \times N$, analogue au *avg-pooling*, également sans réduction de résolution.
- `mm.readImg(h, w, dtype='float')` — utile pour lire la carte d'entrée en virgule flottante.

10.10.2.5 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entiers H et W.
- H lignes suivantes : W valeurs réelles chacune.
- Ligne suivante : Entiers k et s.
- Ligne suivante : `max` ou `avg`.

Sortie :

- Ligne 1 : Entiers O_h et O_w.
- O_h lignes suivantes : O_w valeurs réelles chacune, avec 4 décimales.

10.10.2.6 📌 Exemples

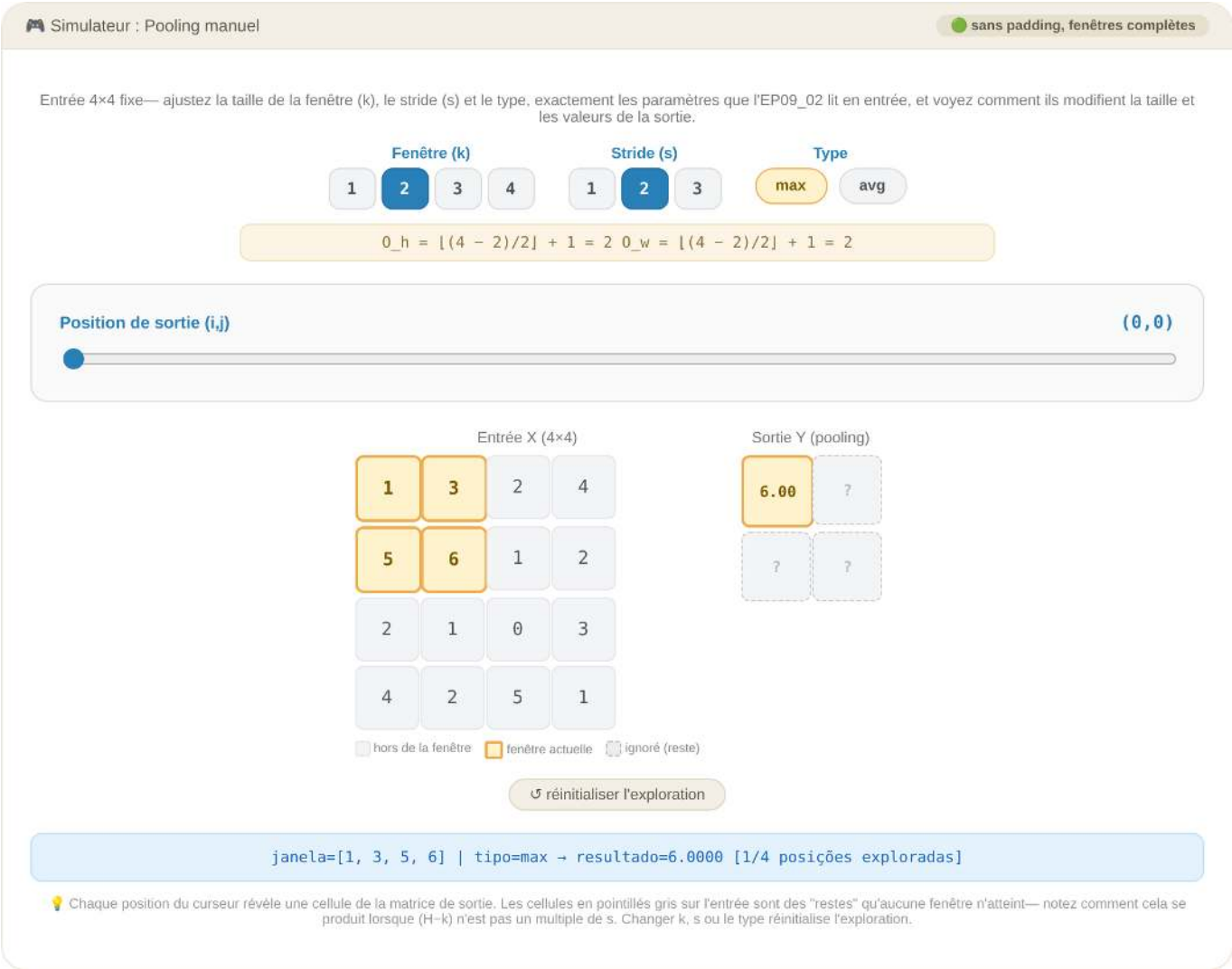
Entrée	Sortie	Observation
4 4 1 3 2 4 5 6 1 2 2 1 0 3 4 2 5 1 2 2 max	2 2 6.0000 4.0000 4.0000 5.0000	*Pooling* maximum, fenêtre 2×2, *stride* 2.
4 4 1 3 2 4 5 6 1 2 2 1 0 3 4 2 5 1 2 2 avg	2 2 3.7500 2.2500 2.2500 2.2500	*Pooling* moyen sur les mêmes fenêtres.

```
%%writefile EP09_02.py
# Code 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.fr/cap09/sim-ep0902.html

Figure 10.44: Simulateur EP09_02 : Pooling manuel (maximum vs. moyenne, avec fenêtre k et pas s réglables)

10.10.3 EP09_03 🟡 Comptage des paramètres entraînables d'un CNN

Cet EP formalise le comptage des paramètres entraînables d'un *CNN*. Étant donné la description textuelle d'une petite architecture, composée de couches convolutives, de *pooling* et entièrement connectées, déterminez, pour chaque couche, le nombre de paramètres entraînables et le total du réseau.

L'architecture doit être interprétée **séquentiellement** : la sortie d'une couche convolutive devient l'entrée de la couche suivante compatible. Ainsi, le nombre de canaux produits par une couche `CONV` détermine le nombre de canaux d'entrée (`cin`) de la couche convolutive suivante.

Dans une couche convolutive, il est important de distinguer **les canaux d'entrée** et **les canaux de sortie** :

- c_{in} (*channels in*) est le nombre de **canaux qui entrent dans la couche**. Une image en niveaux de gris possède $c_{in} = 1$, alors qu'une image RVB possède $c_{in} = 3$. Dans une couche convolutive intermédiaire, `cin` est généralement égal au nombre de canaux produits par la couche `CONV` précédente.
- c_{out} (*channels out*) est le nombre de **canaux produits par la couche**. Il est égal au nombre de filtres utilisés. Par conséquent, si une couche possède 16 filtres, elle produit $c_{out} = 16$ canaux.

Par exemple, considérez la séquence :

```
CONV 3 3 1 8 1
POOL
CONV 3 3 8 16 1
POOL
FC 784 10 1
```

La première convolution reçoit une image avec un canal et produit 8 canaux. Après le *pooling*, la seconde convolution reçoit ces 8 canaux et produit 16 canaux. La couche `POOL` ne modifie pas le nombre de canaux, elle peut seulement réduire les dimensions spatiales. La couche `FC` reçoit le nombre d'entrées indiqué dans la description elle-même.

Chaque filtre convolutif possède des dimensions

$$k_h \times k_w \times c_{in}.$$

Ainsi, une couche avec c_{out} filtres possède

$$k_h \cdot k_w \cdot c_{in} \cdot c_{out}$$

poids. S'il y a un biais, on ajoute un paramètre pour chaque filtre, totalisant ainsi c_{out} paramètres supplémentaires.

Le point central de cet exercice est d'observer que la quantité de paramètres d'une couche convolutive **ne dépend pas des dimensions spatiales** ($H \times W$) de la carte de caractéristiques. Cela est dû au **partage des poids** : le même filtre est réutilisé à différentes positions de l'entrée.

10.10.3.1 📋 Directives d'implémentation

1. **Entrée :** Lire l'entier L (nombre de couches de l'architecture, dans l'ordre où elles sont appliquées).
2. **Couches :** Lire L lignes, chacune décrivant une couche dans l'un des trois formats :
   - `CONV kh kw cin cout bias` — couche convolutive avec un *kernel* $k_h \times k_w$, c_{in} canaux d'entrée, c_{out} canaux de sortie et `bias` (0 ou 1), indiquant s'il y a un biais par filtre ;
   - `POOL` — couche de *pooling* (maximum ou moyen), qui ne possède pas de paramètres entraînables et préserve le nombre de canaux ;
   - `FC in out bias` — couche entièrement connectée avec `in` entrées, `out` sorties et `bias` (0 ou 1), indiquant s'il y a un biais par neurone.

3. **Cohérence entre les couches `CONV` :** dans une séquence de couches convolutives, le `cin` d'une couche doit correspondre au `cout` de la couche convolutive précédente. Une couche `POOL` ne modifie pas ce nombre de canaux.

   Par exemple :

   ```
   CONV 3 3 1 8 1
   POOL
   CONV 3 3 8 16 1
   ```

   La première `CONV` produit 8 canaux, qui sont reçus par la seconde `CONV`. Par conséquent, dans la seconde couche, `cin=8` et `cout=16`.

4. **Paramètres d'une couche `CONV` :**

   Chacun des c_{out} filtres possède $k_h \cdot k_w \cdot c_{in}$ poids. Par conséquent,

$$P_{\text{CONV}} = k_h \cdot k_w \cdot c_{in} \cdot c_{out} + c_{out} \cdot \text{bias}.$$

5. **Paramètres d'une couche `FC` :**

$$P_{\text{FC}} = \text{in} \cdot \text{out} + \text{out} \cdot \text{bias}.$$

6. **Paramètres d'une couche `POOL` :** toujours 0.
7. **Total du réseau :** additionner les paramètres entraînables de toutes les couches.
8. **Sortie :** Pour chaque couche, dans l'ordre de lecture, imprimer `Camada i: P`, où i commence à 1 et P est le nombre de paramètres de cette couche. À la fin, imprimer `Total: T`.

10.10.3.2 📐 Exemple pour comprendre `cin` et `cout`

Considérez la séquence :

```
CONV 3 3 1 8 1
POOL
CONV 3 3 8 16 1
```

Dans la première couche :

- `cin=1` : un canal entre ;
- `cout=8` : il existe 8 filtres et, par conséquent, 8 canaux sortent.

Chaque filtre possède

$$3 \cdot 3 \cdot 1 = 9$$

poids. Comme il existe 8 filtres :

$$9 \cdot 8 = 72$$

poids. Avec un biais par filtre :

$$72 + 8 = 80.$$

Dans la seconde couche :

- `cin=8` : les 8 canaux produits par la première `CONV` entrent ;
- `cout=16` : il existe 16 filtres et, par conséquent, 16 canaux sortent.

Chaque filtre possède

$$3 \cdot 3 \cdot 8 = 72$$

poids. Comme il existe 16 filtres :

$$72 \cdot 16 = 1152$$

poids. Avec 16 biais :

$$1152 + 16 = 1168.$$

Ainsi, les deux couches possèdent, respectivement, **80** et **1168 paramètres entraînables**.

Observez que `cout` **n'est pas** *cin* multiplié par le nombre de filtres. Le nombre de filtres est exactement `cout` : chaque filtre combine tous les canaux d'entrée et produit **un seul canal de sortie**.

10.10.3.3 📌 Contraintes computationnelles

- **Indépendance de la dimension spatiale :** l'entrée ne fournit pas $H \times W$. Le comptage d'une couche `CONV` dépend uniquement de `kh`, `kw`, `cin` et `cout`.
- **Cohérence des canaux :** pour deux couches `CONV` consécutives, le `cin` de la seconde doit être égal au `cout` de la première. Une couche `POOL` préserve le nombre de canaux.
- **`bias` toujours 0 ou 1 :** multiplier directement le terme de biais par cette valeur.
- **Couches `POOL` sans arguments supplémentaires :** la ligne contient uniquement le mot `POOL`.
- **Couches `FC` :** le nombre d'entrées `in` est fourni explicitement. Il n'est pas nécessaire de calculer les dimensions spatiales produites par les couches précédentes.
- Toutes les valeurs numériques d'entrée sont des entiers non négatifs.

10.10.3.4 🧠 Fondement théorique

Élément	Rôle dans le comptage des paramètres
c_{in}	Nombre de canaux reçus par la couche
c_{out}	Nombre de filtres et, par conséquent, de canaux produits par la couche
Filtre convolutif	Chaque filtre possède $k_h \cdot k_w \cdot c_{in}$ poids et produit un canal de sortie
Partage des poids	Le même filtre est réutilisé à différentes positions de l'entrée, rendant le comptage indépendant de $H \times W$
Biais	Un seul paramètre supplémentaire par filtre (`CONV`) ou par neurone (`FC`)
Pooling	Peut modifier $H \times W$, mais ne possède pas de paramètres entraînables et préserve le nombre de canaux
Couche `FC`	Possède un poids pour chaque combinaison entre entrée et neurone de sortie

10.10.3.5 🧩 Méthodes de `morph.py` pouvant aider

Cet exercice est purement arithmétique et n'utilise pas directement les fonctions de `morph.py`. Le comptage peut, cependant, être vérifié sur une architecture réelle implémentée en PyTorch via :

```
sum(p.numel() for p in modelo.parameters())
```

Cette expression comptabilise les paramètres du modèle, y compris les poids et les biais.

10.10.3.6 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entier L.
- L lignes suivantes : description de chaque couche, au format `CONV kh kw cin cout bias`, `POOL` ou `FC in out bias`.

Sortie :

- L lignes au format `Camada i: P`.
- Dernière ligne : `Total: T`.

10.10.3.7 📌 Exemples

Entrée	Sortie	Observation
3 CONV 3 3 1 8 1 POOL FC 1352 10 1	Camada 1: 80 Camada 2: 0 Camada 3: 13530 Total: 13610	Réseau simple avec une convolution, *pooling* et couche de classification.
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	Petit CNN avec deux convolutions, deux *poolings* et une couche entièrement connectée.
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	Petit CNN avec deux convolutions, *pooling* intermédiaire et deux couches entièrement connectées pour la classification.

```
%%writefile EP09_03.py
# Code Python
```

```
Overwriting EP09_03.py
```

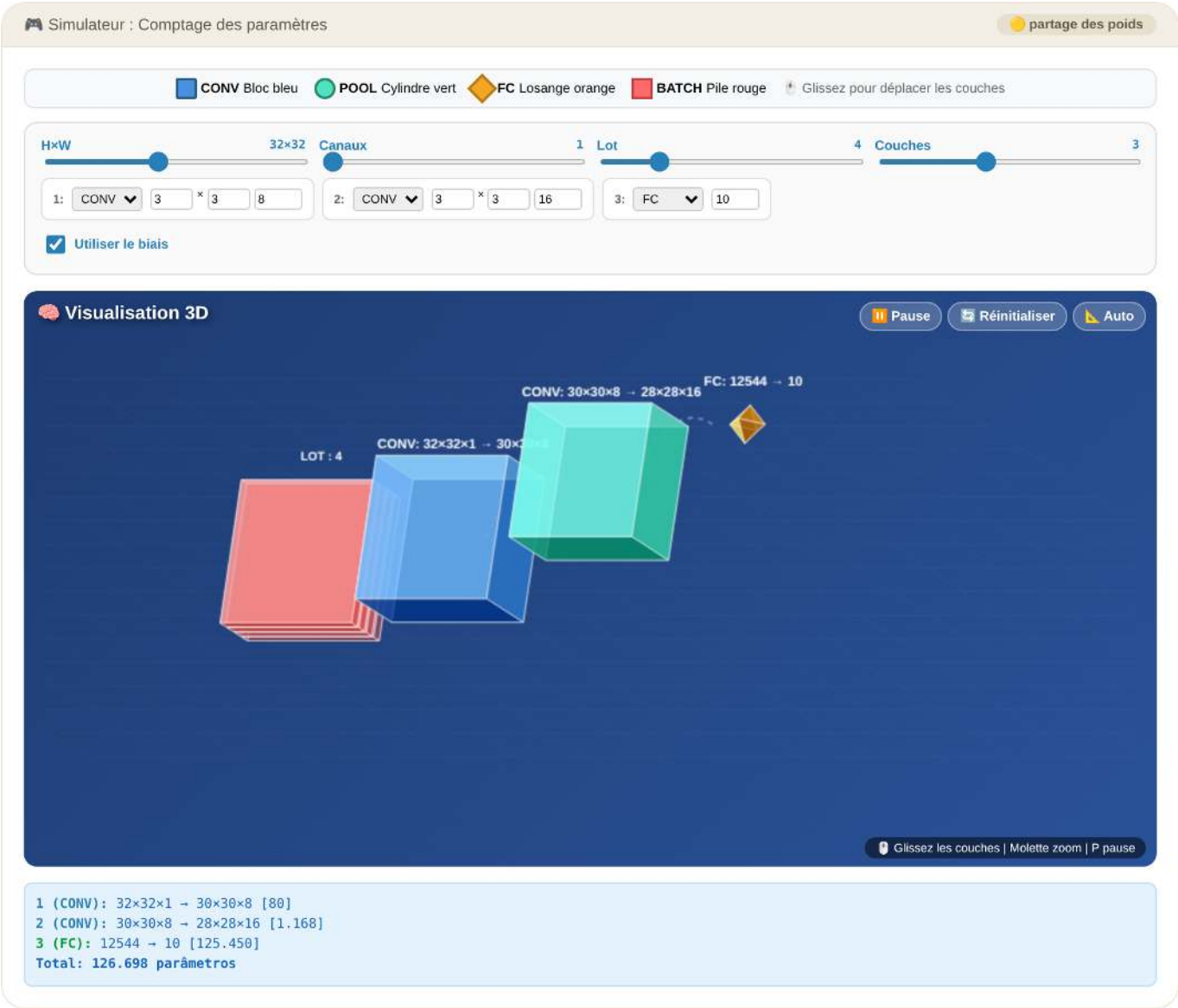
```
TestSuite("EP09_03.py").run()
```

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

10.10.4 EP09_04 🟡 Intersection sur Union (IoU) et Suppression des Non-Maximums (NMS)

Les modèles de détection d'objets peuvent produire **plusieurs boîtes englobantes candidates** pour un même objet, avec différentes positions et scores de confiance. L'étape de post-traitement chargée d'éliminer ces détections redondantes est la **Suppression des Non-Maximums (NMS)**, dont l'opération fondamentale utilise la métrique de l'**Intersection sur Union (IoU)**.

La NMS utilise cette mesure pour décider quelles boîtes doivent être conservées. En général, la boîte avec la plus haute confiance est sélectionnée en premier ; ensuite, les boîtes qui présentent une IoU supérieure à un certain seuil avec la boîte sélectionnée sont considérées comme redondantes et supprimées. Le processus est répété jusqu'à ce qu'il ne reste plus de boîtes candidates.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap09/sim-ep0903.html

Figure 10.45: Simulateur EP09_03 : Comptage de paramètres — Convolution vs. couche entièrement connectée

Dans cet exercice, vous devrez implémenter l'algorithme de NMS à partir de zéro, en calculant l'IoU entre les boîtes et en appliquant successivement le critère de sélection et de suppression pour produire l'ensemble final de détections.

10.10.4.1 📋 Directives d'implémentation

1. **Entrée :** Lire l'entier N (nombre de boîtes candidates) et le seuil réel τ (seuil d'IoU pour la suppression), sur la même ligne.
2. **Boîtes :** Lire N lignes, chacune avec cinq valeurs réelles :

   ```
   x1 y1 x2 y2 score
   ```

   où (x_1, y_1) représente le coin supérieur gauche, (x_2, y_2) le coin inférieur droit et `score` le score de confiance.
3. **Intersection sur Union :** Pour deux boîtes A et B,

   $$IoU(A, B) = \frac{\text{Aire}(A \cap B)}{\text{Aire}(A \cup B)}.$$

   L'aire d'intersection doit être calculée à partir du chevauchement des intervalles en x et en y. S'il n'y a pas de chevauchement, l'aire d'intersection est nulle.
4. **Algorithme glouton de NMS :**
   a. Trier les boîtes par `score` décroissant. En cas d'égalité, conserver l'ordre original de lecture.
   b. Sélectionner la boîte avec le score le plus élevé parmi les boîtes restantes et l'ajouter à l'ensemble de sortie.
   c. Calculer l'IoU entre la boîte sélectionnée et **toutes les boîtes encore restantes**. Supprimer les boîtes pour lesquelles

   $$\text{IoU} > \tau.$$

   d. Répéter les étapes (b) et (c) jusqu'à ce qu'il ne reste plus de boîtes.
5. **Sortie :** Pour chaque boîte conservée, dans l'ordre où elle a été sélectionnée, imprimer son indice original (position de lecture, commençant à 0) et son `score`, formaté avec 4 décimales. À la fin, imprimer :

   ```
   Total conservées : X
   ```

10.10.4.2 📌 Contraintes de calcul

- **Suppression stricte :** seules les boîtes avec IoU $> \tau$ sont supprimées. Les boîtes avec IoU $= \tau$ sont conservées.
- **Indices originaux :** la sortie fait référence à la position où chaque boîte a été lue dans l'entrée (commençant à 0), et non à sa position après le tri.
- **Tri stable :** en cas de `score` égaux, l'ordre original de lecture doit être préservé.
- **Rectangles alignés sur les axes :** toutes les boîtes sont spécifiées par deux coins, avec $x_1 < x_2$ et $y_1 < y_2$ garantis en entrée.
- **Coordonnées et scores :** les valeurs réelles peuvent être positives ou négatives, selon les limites définies par l'entrée, mais les dimensions des boîtes sont toujours positives.

10.10.4.3 🧠 Fondements théoriques

Élément	Rôle dans le post-traitement de détection
IoU	Quantifie le chevauchement spatial entre deux boîtes ; IoU = 1 pour des boîtes identiques et IoU = 0 pour des boîtes sans chevauchement
Tri par confiance	Fait en sorte que la boîte avec le plus grand `score` soit analysée en premier
Seuil τ	Définit la quantité de chevauchement nécessaire pour qu'une boîte soit considérée comme redondante
Suppression	Supprime les boîtes qui présentent un grand chevauchement avec une boîte déjà sélectionnée
Boîtes distantes	Ont une IoU proche de zéro et, en général, ne sont pas supprimées par cette règle

10.10.4.4 🧩 Méthodes de `morph.py` qui peuvent aider

- `mm.IoU(boxA, boxB)` — calcule la métrique d'IoU, mais attend les boîtes au format (x, y, w, h), c'est-à-dire coin supérieur gauche, largeur et hauteur. L'entrée de cet exercice utilise le format (x_1, y_1, x_2, y_2). La conversion est directe :

$$w = x_2 - x_1, \qquad h = y_2 - y_1.$$

  L'utilisation de cette fonction est facultative. L'objectif principal de l'exercice est d'implémenter correctement le processus de sélection et de suppression de la NMS.

10.10.4.5 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : entier N et réel τ.
- N lignes suivantes : x_1 y_1 x_2 y_2 score.

Sortie :

- Une ligne par boîte conservée, dans l'ordre de sélection : `indice score`.
- Dernière ligne : `Total conservées : X`.

```
%%writefile EP09_04.py
# Code Python
```

```
Overwriting EP09_04.py
```

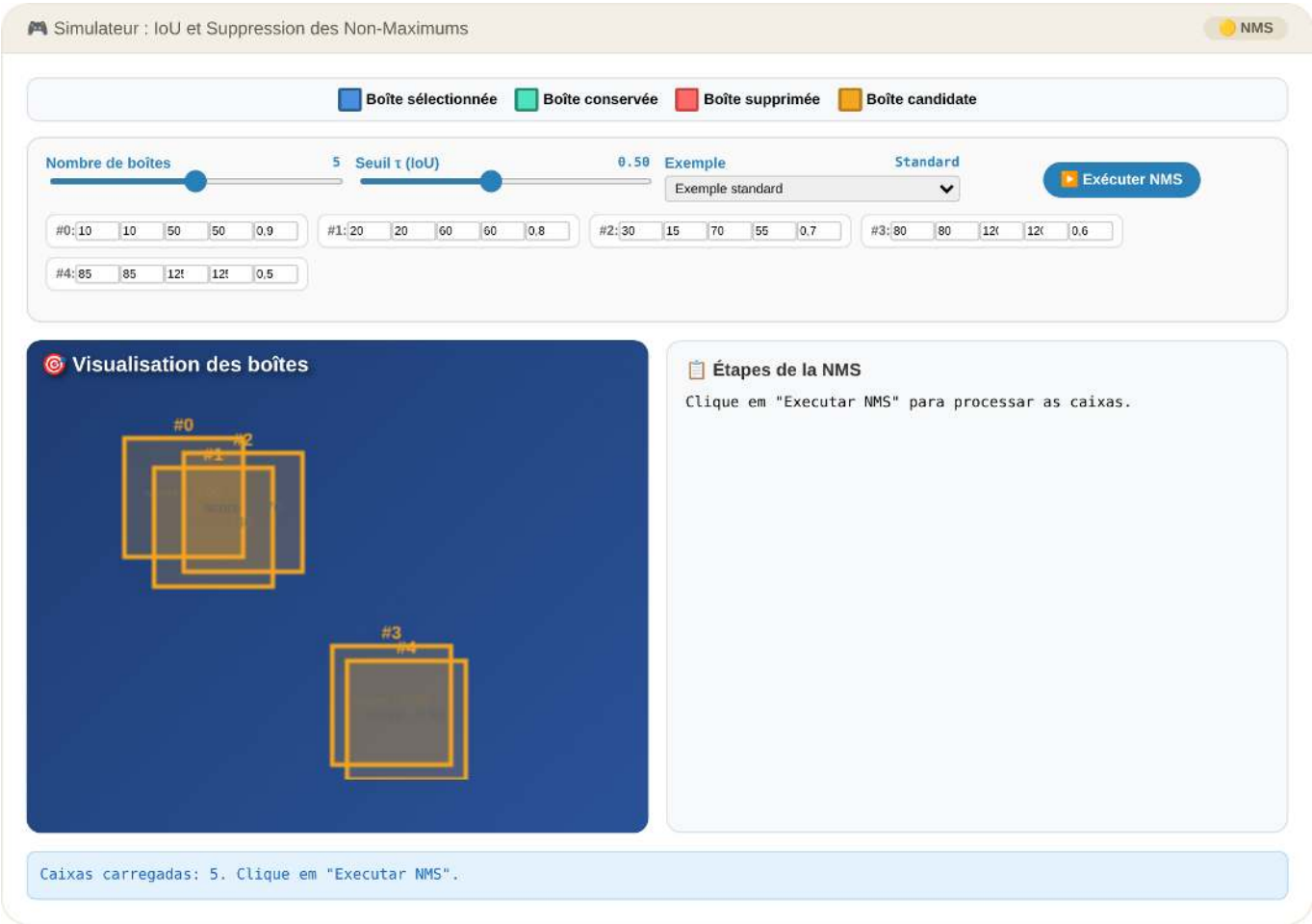
```
TestSuite("EP09_04.py").run()
```

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

10.10.5 EP09_05 🟠 Évaluation de segmentation : IoU et Dice pixel par pixel

Le Bloc 2 de la section « Segmentation sémantique avec l'architecture U-Net » définit, en quelques lignes, la fonction `iou_mascaras`, utilisée pour mesurer la qualité de la ligne de base morphologique classique (lissage + Otsu + ouverture) et, plus loin, celle de la U-Net elle-même après entraînement. Contrairement à l'IoU de l'EP09_04 — calculé sur des **boîtes englobantes** (régions rectangulaires décrites par quatre nombres) —, l'IoU de segmentation est calculé **pixel par pixel** : chaque position de l'image est comparée individuellement entre le masque prédit et le masque de référence.

Vous êtes chargé de généraliser cette évaluation, en implémentant non seulement l'IoU pixel par pixel, mais aussi le **coefficient de Dice**, une autre métrique de chevauchement largement utilisée en segmentation



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap09/sim-ep0904.html

Figure 10.46: Simulateur EP09_04 : IoU et suppression des non-maximums (NMS)

médicale (y compris dans la fonction `perda_dice`, mentionnée dans le même bloc du chapitre comme base de la fonction de perte utilisée pour entraîner la U-Net).

10.10.5.1 📋 Directives d'implémentation

1. **Entrée :** Lire les dimensions $H \times W$ des masques.
2. **Masque prédit :** Lire H lignes contenant chacune W valeurs entières (0 ou 1) — par exemple, la sortie d'une U-Net après seuillage à 0,5 sur la sigmoïde, comme dans le Bloc 4 du chapitre.
3. **Masque de référence :** Lire H lignes supplémentaires avec chacune W valeurs entières (0 ou 1) — la *vérité terrain* (*ground truth*).
4. **Intersection et union :** En considérant chaque pixel comme appartenant à l'objet lorsque sa valeur est non nulle,
$$\text{intersection} = \sum_{i,j} \mathbb{1}[P_{ij} = 1 \wedge R_{ij} = 1], \qquad \text{union} = \sum_{i,j} \mathbb{1}[P_{ij} = 1 \vee R_{ij} = 1].$$
5. **IoU pixel par pixel :**
$$\text{IoU} = \frac{\text{intersection}}{\text{union}}.$$
6. **Coefficient de Dice :**
$$\text{Dice} = \frac{2 \cdot \text{intersection}}{|P| + |R|},$$
où $|P|$ et $|R|$ sont le nombre total de pixels d'objet dans chaque masque.
7. **Convention pour les masques vides :** si **les deux** masques ne contiennent aucun pixel d'objet (union $= 0$ et $|P| + |R| = 0$), considérez la correspondance comme trivialement parfaite : IoU = Dice = 1,0.
8. **Sortie :** Deux lignes, `IoU: X.XXXX` et `Dice: X.XXXX`, chaque valeur avec 4 décimales.

10.10.5.2 📌 Contraintes de calcul

- **Toute valeur non nulle compte comme objet :** traitez les valeurs différentes de 0 (pas seulement 1) comme appartenant au masque, en répliquant la vérification `predita > 0` utilisée dans `iou_mascaras` au chapitre.
- **Mêmes dimensions :** les deux masques ont toujours exactement $H \times W$ éléments.
- **Convention de vide :** appliquez la règle de l'item 7 **uniquement** lorsque les deux masques sont entièrement vides ; si un seul est vide, l'intersection est 0 et l'IoU/Dice résultant sera également 0.

10.10.5.3 🧠 Fondement théorique

Élément	Rôle dans l'évaluation de segmentation
IoU pixel par pixel	Généralise la métrique de l'EP09_04 à des régions de forme arbitraire — pas seulement des rectangles — en comparant le masque prédit et la référence position par position
Coefficient de Dice	Métrique liée à l'IoU (toujours Dice $\geq$ IoU), plus sensible aux petites intersections et largement utilisée comme fonction de perte en segmentation (fonction `perda_dice` du chapitre)
Convention des masques vides	Évite la division par zéro et reconnaît que « aucun objet prédit, aucun objet réel » est, par définition, une réussite

Élément	Rôle dans l'évaluation de segmentation
Comparaison classique vs. U-Net	Le chapitre utilise exactement ce type de métrique pour justifier, numériquement, pourquoi la U-Net surpasse la ligne de base morphologique dans des scénarios à faible contraste

10.10.5.4 🧩 Méthodes de `morph.py` pouvant aider

- `mm.readImg(h, w, dtype='uint8')` — lit directement chaque masque binaire $h \times w$ depuis l'entrée standard (les valeurs 0/1 tiennent parfaitement dans le type entier standard).
- La fonction `iou_mascaras` elle-même, définie dans le Bloc 2 de la section U-Net du chapitre (ne fait pas partie de `morph.py`, mais du code du chapitre), est l'inspiration directe de cet exercice — il vaut la peine de relire ces quelques lignes avant de programmer.
- Pour une extension optionnelle (non exigée par cet EP), `mm.connectedComponents` ou `mm.label0` (vus dans le contexte de l'analyse des composantes connexes) permettraient de labelliser chaque nodule individuellement et de calculer l'IoU **par composante**, plutôt que sur le masque entier.

10.10.5.5 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Entiers H et W.
- H lignes suivantes : W valeurs entières (0 ou 1) — masque prédit.
- H lignes suivantes : W valeurs entières (0 ou 1) — masque de référence.

Sortie :

- Ligne 1 : `IoU: X.XXXX`.
- Ligne 2 : `Dice: X.XXXX`.

💡 Exemple illustratif

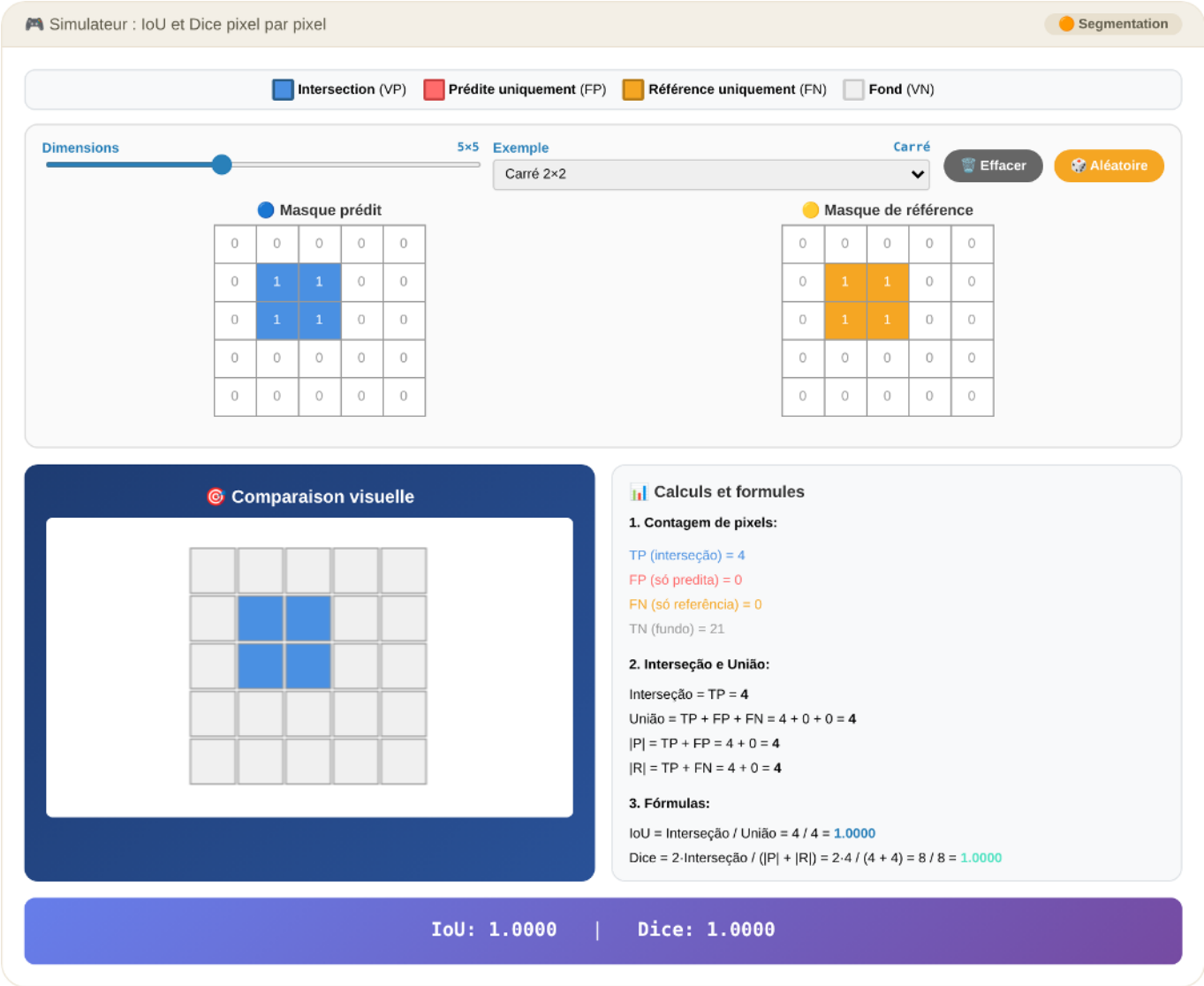
Considérons un masque prédit avec un carré 2×2 de pixels actifs et une référence décalée d'une colonne, se chevauchant sur seulement la moitié de la surface :

```
Prédit          Référence
0 0 0 0         0 0 0 0
0 1 1 0         0 0 1 1
0 1 1 0         0 0 1 1
0 0 0 0         0 0 0 0
```

Intersection = 2 pixels, union = 6 pixels $(4+4-2)$, donc IoU $= 2/6 \approx 0{,}3333$ et Dice $= 2 \cdot 2/(4+4) = 0{,}5000$ — notez que le Dice est toujours supérieur ou égal à l'IoU pour le même chevauchement.

10.10.5.6 📌 Exemples

Entrée	Sortie	Observation
4 4 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0	IoU: 0.3333 Dice: 0.5000	Masques 4×4 avec chevauchement partiel de 2 pixels.



Versão interativa: https://fzampirolli.github.io/pdi-vc/simuladores/py.fr/cap09/sim-ep0905.html

Figure 10.47: Simulador EP09_05: Évaluation de la segmentation — IoU et Dice pixel par pixel

```
%%writefile EP09_05.py
# Code Python
```

```
Overwriting EP09_05.py
```

```
TestSuite("EP09_05.py").run()
```

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

10.10.6 EP09_06 🔴 *Pipeline* intégré : de la détection à la mesure du monde réel

Cet exercice final intègre les deux exercices de détection et le principe de **photogrammétrie** présenté dans la section « Photogrammétrie et référence d'échelle » — exactement le même calcul que celui implémenté dans la figure de mesure par référence d'échelle de ce chapitre. Le scénario reproduit une situation réaliste : un détecteur (Faster R-CNN ou YOLO) génère **plusieurs boîtes candidates se chevauchant** pour le même objet d'intérêt ; après les avoir filtrées par NMS, la boîte survivante de plus haute confiance est utilisée, avec une boîte de référence de largeur réelle connue (comme la carte de 8,56 cm), pour estimer les dimensions réelles de l'objet détecté.

10.10.6.1 📋 Directives d'implémentation

1. **Référence connue :** Lire la valeur réelle L_{ref} (largeur réelle de l'objet de référence, en cm), puis les quatre réels x_1 y_1 x_2 y_2 de sa boîte englobante en pixels (déjà connue, sans nécessité de détection).
2. **Candidates de l'objet à mesurer :** Lire l'entier N (nombre de boîtes candidates produites par le détecteur pour l'objet d'intérêt) et le seuil réel τ ; puis lire les N lignes de boîtes candidates, chacune avec x_1 y_1 x_2 y_2 score.
3. **Étape 1 — NMS :** Appliquer exactement l'algorithme de Suppression Non Maximale de l'EP09_04 aux N boîtes candidates, en utilisant le seuil τ, pour éliminer les détections redondantes du même objet.
4. **Étape 2 — Sélection de la boîte finale :** Après le NMS, la boîte de plus grand `score` parmi celles conservées est la détection finale de l'objet (l'entrée garantit que toutes les boîtes candidates correspondent à un seul objet physique, donc la première boîte sélectionnée par le NMS est déjà le résultat final).
5. **Étape 3 — Mesure par référence d'échelle :** Calculer le rapport cm/pixel = L_{ref}/largeur de la référence en p et l'appliquer à la fois à la largeur et à la hauteur (en pixels) de la boîte finale de l'objet, obtenant ainsi ses dimensions réelles estimées en centimètres.
6. **Sortie :** D'abord, une ligne par boîte conservée après le NMS (même format que l'EP09_04) : `indice score`. Ensuite, la ligne `Total conservées : X`. Enfin, la ligne `Objet : L x A cm`, où L et A sont la largeur et la hauteur estimées de l'objet, chacune avec 2 décimales.

10.10.6.2 📌 Contraintes de calcul

- **Réutiliser intégralement le NMS de l'EP09_04** — même règle de départage, même critère de suppression (IoU $> \tau$).
- **La référence ne passe pas par le NMS :** sa boîte est donnée directement, sans candidates concurrentes.
- **Rapport unique pour la largeur et la hauteur :** comme dans la figure de photogrammétrie du chapitre, le même rapport cm/pixel (dérivé de la largeur de la référence) est appliqué à la fois à la largeur et à la hauteur de l'objet — il n'y a pas de calibration verticale séparée.

10.10.6.3 🧠 Fondement théorique

Étape	Concept du chapitre
Plusieurs boîtes candidates	Sortie brute d'un détecteur comme le Faster R-CNN ou le YOLO, avant le post-traitement
NMS (EP09_04)	Filtre les détections redondantes, ne conservant que la plus fiable pour l'objet
Référence d'échelle connue	Même principe que la carte de 8,56 cm utilisée dans la section « Photogrammétrie et référence d'échelle »

Étape	Concept du chapitre
Conversion pixel → centimètre	Règle de trois simple : cm/pixel = L_{ref}/w_{ref_px}, appliquée à la boîte finale de l'objet

10.10.6.4 🧩 Méthodes de `morph.py` pouvant aider

- `mm.IoU(boxA, boxB)` — la même fonction suggérée dans l'EP09_04, ici réutilisée dans l'étape de NMS de ce *pipeline* intégré (rappelez-vous de la conversion de format : $w = x_2 - x_1$, $h = y_2 - y_1$).
- Si vous avez déjà résolu l'EP09_04 en encapsulant le NMS dans une fonction dédiée, c'est le moment idéal de **réutiliser ce code** — l'intégration de modules déjà testés individuellement est exactement la pratique d'ingénierie que cet exercice veut renforcer.

10.10.6.5 📦 Spécification d'entrée et de sortie (VPL)

Entrée :

- Ligne 1 : Réel L_{ref}.
- Ligne 2 : x_1 y_1 x_2 y_2 de la boîte de référence.
- Ligne 3 : Entier N et réel τ.
- N lignes suivantes : x_1 y_1 x_2 y_2 score des boîtes candidates de l'objet.

Sortie :

- Une ligne par boîte conservée après le NMS : `indice score`.
- Ligne suivante : `Total conservées : X`.
- Dernière ligne : `Objet : L x A cm`.

10.10.6.6 📌 Exemples

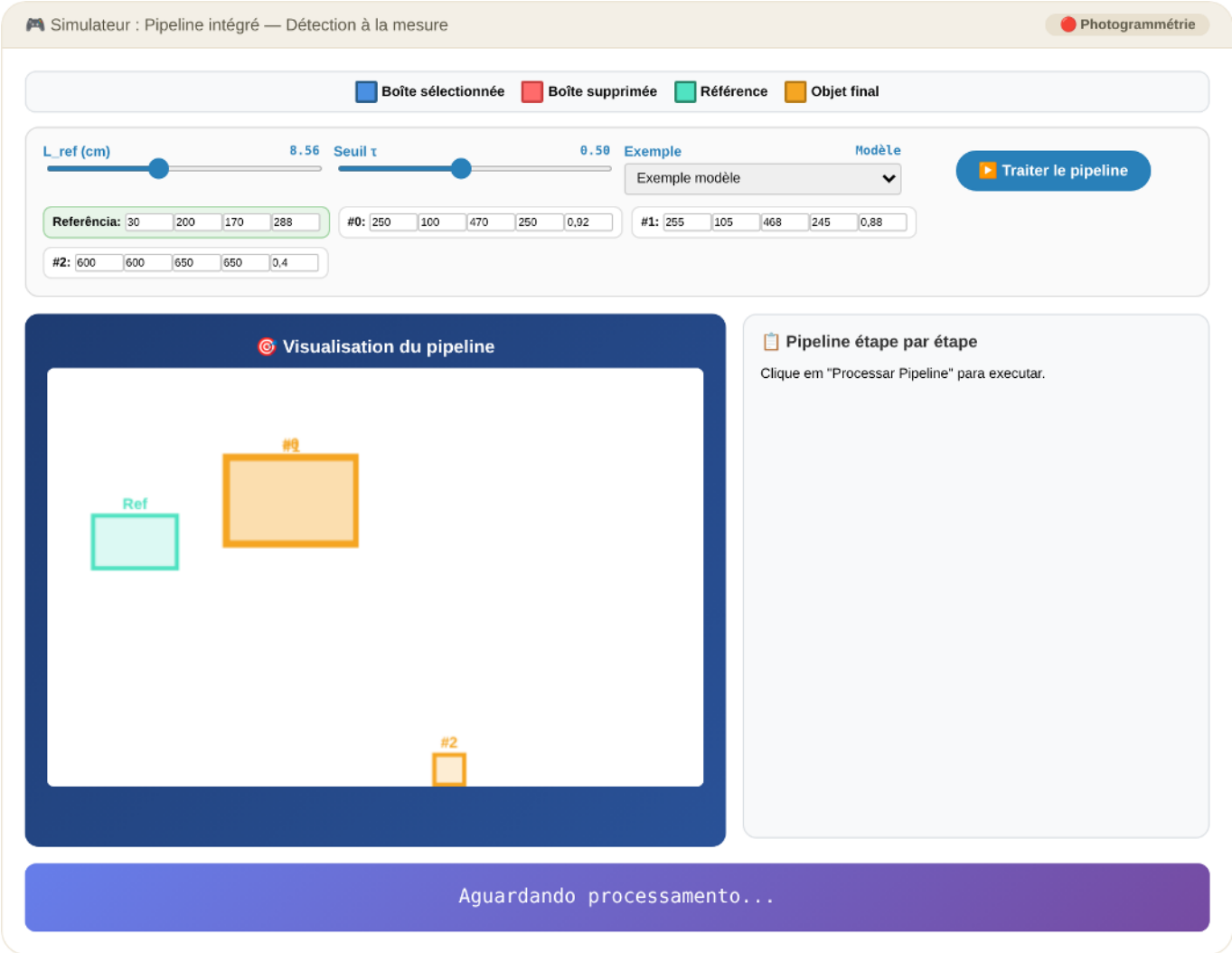
Entrée	Sortie	Observation
8.56 30 200 170 288 3 0.5 250 100 470 250 0.92 255 105 468 245 0.88 600 600 650 650 0.40	0 0.9200 2 0.4000 Total conservées : 2 Objet : 13.45 x 9.17 cm	La boîte 1 est supprimée car elle chevauche fortement la boîte 0 ; la détection finale de l'objet est la boîte 0.

```
%%writefile EP09_06.py
# Code 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.fr/cap09/sim-ep0906.html

Figure 10.48: Simulateur EP09_06 : Pipeline Intégré — Détection à la Mesure du Monde Réel

Références

BAHDANAU, Dzmitry; CHO, Kyunghyun; BENGIO, Yoshua. Neural Machine Translation by Jointly Learning to Align and Translate. *In*: 2015. Disponível em: <https://arxiv.org/abs/1409.0473>

BARRERA, Junior *et al.* MMach: a mathematical morphology toolbox for the KHOROS system. **Journal of Electronic Imaging**, v. 7, n. 1, p. 174-210, 1998.

DOUGHERTY, Edward R.; LOTUFO, Roberto A. **Hands-on morphological image processing**. *[S.l.]*: SPIE press, 2003. v. 59

KNUTH, Donald E. Literate Programming. **The Computer Journal**, v. 27, n. 2, p. 97-111, 1984.

LOTUFO, Roberto A. *et al.* MMachLib functions and MMach operators. *In*: 1997.

MATHERON, Georges. **Random Sets and Integral Geometry**. New York: John Wiley & Sons, 1975.

VIOLA, Paul; JONES, Michael. Rapid Object Detection Using a Boosted Cascade of Simple Features. *In*: IEEE Computer Society, 2001.

ZAMPIROLLI, Francisco A. **MCTest: Como Criar e Corrigir Exames Parametrizados**. *[S.l.]*: Independente, 2023.

ZAMPIROLLI, Francisco de Assis *et al.* A Practical Digital Image Processing Course with morph.py. *In*: Porto Alegre, RS, Brasil: Sociedade Brasileira de Computação (SBC), 2024. Disponível em: <https://sol.sbc.org.br/index.php/educomp/article/view/28199>

ZAMPIROLLI, Francisco de Assis *et al.* Teaching Hands-On Digital Image Processing with morph.py: Methods and Comprehensive Results. **Revista Brasileira de Informática na Educação**, v. 33, p. 990-1026, 2025.

ZAMPIROLLI, Francisco de Assis *et al.* Intelligent Feedback for Individualized Introductory Programming Exercises. **Computer Applications in Engineering Education**, v. 34, n. 1, p. e70132, 2026.

Annexe A

À propos de MCTest

MCTest est un système *open source* pour la création et la correction automatique d'examens paramétrés (Zampirolli, 2023). Il prend en charge les questions à choix multiples, dissertatives et de programmation, ces dernières intégrées au VPL de Moodle, décrit à l'Annexe B.

La version de MCTest utilisée dans ce livre est la **5.4**, disponible sur https://github.com/fzampirolli/mctest. Le sous-dossier `book` du dépôt contient les versions **5.3**, décrite dans (Zampirolli, 2023), et **5.4**, utilisée pour générer les épreuves présentées dans ce livre et actuellement en production à l'UFABC (https://mctest.ufabc.edu.br).

A.1 Installation de MCTest

L'installation recommandée utilise VirtualBox avec Ubuntu 22.04. Dans le terminal Ubuntu, en tant que *root*, exécutez :

```
wget https://raw.githubusercontent.com/fzampirolli/mctest/master/_setup-all.sh
```

Remplacez ensuite `yourLogin` par le nom d'utilisateur local et exécutez :

```
sed -i 's/\/home\/fz\//\/home\/yourLogin\//g' _setup-all.sh
source _setup-all.sh
pip install mysqlclient
```

Après quelques minutes, MCTest sera configuré. Pour l'exécuter :

```
source /home/yourLogin/PycharmProjects/runDjango.sh
```

Le système est alors accessible à l'adresse `http://127.0.0.1:8000`.

A.2 Créer un examen dans MCTest

Dans MCTest, chaque examen est configuré sur un écran **Exam**, qui réunit les classes, les sujets (associés à une matière, comme PDI-VC) et les questions — chaque question appartient à un seul sujet. Outre les attributs habituels de base de données, l'examen possède ses propres paramètres, comme le nombre de questions tirées et le nombre de variantes générées.

Pour les deux examens décrits dans cette annexe, un ensemble de questions paramétrées a été défini par examen, dont un sous-ensemble est tiré aléatoirement pour chaque variante, totalisant **110 variantes distinctes** — une par étudiant inscrit dans les classes.

Le bouton **Create-Variations** génère un nouvel ensemble de variantes à chaque activation. Lorsque les questions sont liées à des activités VPL, l'enseignant reçoit par e-mail un fichier `*linker.json` contenant tous les cas de test des variantes générées. Le bouton **Create-PDF** génère, pour chaque classe, un PDF avec les examens tirés par étudiant, ainsi qu'un fichier `*students_variations.csv` indiquant le nom, l'e-mail et la variante tirée de chaque étudiant.

! Attention

Chaque activation du bouton **Create-Variations** génère un nouvel ensemble de variantes d'examen. Après avoir imprimé le PDF, il est essentiel de **ne pas modifier les attributs de l'examen**, sous peine d'invalider la correction automatique des examens déjà imprimés ou passés.

A.3 Créer une question paramétrique

Une question paramétrique dans MCTest combine trois éléments :

1. **Description** — l'énoncé de la question, écrit en LaTeX, contenant des variables entre `[[code:variable]]`, remplacées par la valeur tirée pour chaque étudiant ;
2. **Bloc de définition** (`[[def: ... ]]`) — un extrait de code Python, intégré à la question elle-même, chargé de tirer les paramètres, de calculer la solution de référence et de construire les cas de test ;
3. **Cas de test pour Moodle** (bloc `moodle_cases`) — un dictionnaire sérialisé en JSON contenant les entrées et sorties attendues, consommé par l'activité VPL.

Un exemple simplifié de définition de question (adapté de la génération d'un échiquier `h × w`) est :

```
[[code:texto]]

\vspace{2mm}\noindent\textbf{Exemple d'entrée :}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_inp]]
\end{verbatim}
\vspace{-2mm}\noindent\textbf{Exemple de sortie :}\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 # DOIT inclure "topic." dans 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

# PARAMÈTRES UTILISÉS DANS L'ÉNONCÉ DE LA QUESTION
height = int(np.random.randint(5, 15))

# ÉNONCÉ COMPLET, AVEC LA VALEUR DE height DÉJÀ INTERPOLÉE
texto = (
    r"\vspace{2mm}\noindent\textbf{Description :}\vspace{-2mm} "
    r"Écrivez un programme qui lit un entier \texttt{W}, représentant la "
    r"largeur (nombre de colonnes) d'un plateau, et affiche ce plateau sous "
    r"la forme d'un échiquier, en utilisant les chiffres \texttt{0} et \texttt{1}. "
    r"Le plateau affiché aura toujours une hauteur (nombre de lignes) égale à "
    f"\\textbf{{{height}}} "
    r"et une largeur égale à la valeur \texttt{W} lue en entrée. "
```

```
    r"En considérant la position $(i, j)$ du plateau (indexée à partir de 0, "
    r"$i$ représentant la ligne et $j$ la colonne), la valeur affichée à "
    r"cette position doit être \texttt{1} si $i + j$ est impair, et \texttt{0} sinon. "
    r"Chaque ligne du plateau doit être affichée sur une ligne séparée, avec "
    r"les valeurs de chaque colonne séparées par un espace."
)

inp_list, out_list, test_cases = [], [], 4
for i in range(test_cases):
    width = int(np.random.randint(500, 1500) / 100)
    inp = str(width) + '\n'
    out = mm.drawImg(chess(height, width)) + '\n'
    inp_list.append(inp)
    out_list.append(out)

cases = {}
cases['skills']      = ["matrices", "boucles", "formatage de la sortie"]
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]
]]
```

La valeur de `height` est tirée une seule fois par variante (pour apparaître dans l'énoncé), tandis que `width` est tirée à chaque cas de test, ce qui augmente la robustesse de la correction automatique.

Il convient de souligner le double rôle de la variable `texto` dans ce mécanisme. Elle est à la fois :

- **le contenu de l'énoncé imprimé**, inséré dans la description LaTeX de la question via la balise `[[code:texto]]`, apparaissant dans le PDF généré par le bouton **Create-PDF** ; et
- **une entrée du dictionnaire exporté vers Moodle**, via la clé `cases['description']`, qui vient composer le JSON `moodle_cases`. C'est précisément ce champ que l'activité VPL affiche à l'étudiant lorsqu'il ouvre la question dans Moodle pour la consulter ou soumettre une solution.

Il existe toutefois une différence importante entre ces deux usages : le PDF est composé en LaTeX et interprète donc normalement des commandes comme `\textbf{}`, `\texttt{}` ou `$...$`. Le VPL de Moodle, lui, **ne** traite **pas** le LaTeX — il attend du texte brut. C'est pourquoi, lors de la construction de `cases['description']`, `texto` n'est pas inséré directement, mais passé par la fonction utilitaire propre à MCTest `latex_to_text()`, qui supprime (ou convertit) les commandes et symboles LaTeX de l'énoncé, produisant une version en texte simple équivalente. Ainsi, `[[code:texto]]` dans le PDF continue de recevoir le `texto` original, avec toute sa mise en forme LaTeX, tandis que `cases['description']` reçoit `latex_to_text(texto)`, garantissant que l'énoncé affiché dans Moodle reste lisible même sans prise en charge du LaTeX.

Comme `texto` est construite à l'intérieur même du bloc `[[def: ...]]` — dans la même portée où `height`, `width` et les autres paramètres sont tirés —, elle est recalculée à chaque variante. Cela garantit que l'énoncé vu par l'étudiant dans Moodle est **toujours identique** à celui imprimé sur l'épreuve papier, même lorsque le texte change d'un étudiant à l'autre avec les valeurs tirées. La Section A.4 reprend ce point sur un cas réel, dans lequel l'énoncé est nettement plus long et décrit, en plus des paramètres numériques, le scénario visuel lui-même généré pour chaque variante.

La Figure A.1 montre la question de l'échiquier réellement générée par MCTest pour l'une des variantes, avec l'énoncé contenant déjà la valeur tirée de `height` et l'exemple d'entrée/sortie correspondant au premier cas de test :

En cliquant sur **Create-PDF** sur l'écran de la question, une nouvelle variante est générée à chaque clic, ce qui permet de vérifier l'énoncé avant de le publier.

1. **Descrição:** Escreva um programa que leia um número inteiro `W`, representando a largura (número de colunas) de um tabuleiro, e imprima esse tabuleiro no formato de um tabuleiro de xadrez, usando os dígitos `0` e `1`. O tabuleiro impresso terá sempre altura (número de linhas) igual a **6** e largura igual ao valor `W` lido da entrada. Considerando a posição (i, j) do tabuleiro (indexada a partir de 0, com i representando a linha e j a coluna), o valor impresso nessa posição deve ser `1` se $i + j$ for ímpar, e `0` caso contrário. Cada linha do tabuleiro deve ser impressa em uma linha separada, com os valores de cada coluna separados por um espaço.

Exemplo de Entrada:

```
9
```

Exemplo de Saída:

```
0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1
```

Figure A.1: Question de l'échiquier générée par MCTest, illustrant l'énoncé paramétré avec la valeur de `height` tirée pour la variante et l'exemple d'entrée/sortie du premier cas de test.

Pour les deux examens du cours, ce mécanisme a été utilisé plus largement : chaque question paramétrique définit non seulement des valeurs numériques, mais aussi, dans certains cas, de petites variations structurelles dans l'énoncé (par exemple, quelle direction cardinale — Nord, Sud, Est, Ouest — doit être évaluée), rendant plus difficile la recherche de solutions toutes faites sur internet ou via des outils d'IA générative.

A.4 Exemple réel : une question du Simulado 4

Pour illustrer le mécanisme décrit dans la section précédente avec un cas concret, voici une question effectivement posée lors du **Simulado 4** (sujet *im4-Opérateurs morphologiques*, difficulté 1, taxonomie de Bloom « se souvenir »), du type question de programmation paramétrée avec intégration Moodle+VPL.

La question demande à l'étudiant d'écrire un programme capable de :

- lire une image binaire, corrompue par un bruit poivre et sel, contenant au moins deux objets géométriques isolés ;
- appliquer un filtrage morphologique (ouverture suivie d'une fermeture) pour éliminer le bruit ;
- extraire, à partir de l'image filtrée, les mesures géométriques de chaque objet (aire, périmètre, centre, boîte englobante, circularité, solidité et nombre de sommets), à l'aide de la fonction auxiliaire `mm.measure` ;
- trier les objets par position (coordonnée X de la boîte englobante, avec Y comme critère de départage) et réattribuer les identifiants séquentiellement ;
- afficher la matrice nettoyée et le tableau des mesures dans le format attendu par le correcteur automatique.

Dans le bloc `[[def: ... ]]` correspondant, un générateur de scène (`gerarCena`) tire aléatoirement la hauteur et la largeur de l'image, le type, la taille et la position de chaque objet (carré, rectangle, triangle ou bloc), en garantissant qu'ils ne se chevauchent pas, puis ajoute un bruit poivre et sel avec une probabilité de 3 % par pixel. Dix cas de test distincts sont ainsi générés pour chaque variante de l'examen, et la paire entrée/sortie du premier cas est réutilisée dans l'énoncé lui-même comme exemple pour l'étudiant. La bibliothèque de morphologie (`mm`) est importée directement depuis `morph.py`, également disponible comme module utilitaire de MCTest.

En résumé, l'énoncé demande à l'étudiant un programme qui :

1. lit, sur la première ligne, deux entiers H et W (hauteur et largeur de l'image) ;
2. lit les H lignes suivantes de la matrice binaire (0 et 1 séparés par des espaces), à l'aide de `mm.readImg(H, W)` ;
3. applique un filtrage morphologique pour supprimer le bruit poivre et sel ;
4. affiche la matrice déjà filtrée, en 0 et 1 séparés par des espaces ;
5. extrait les mesures géométriques des objets avec `mm.measure(imgLimpa)` ;

6. trie les objets et affiche le tableau des mesures dans le format attendu.

L'énoncé apporte encore deux observations importantes pour la correction automatique : (i) l'aire calculée par OpenCV (`cv2.contourArea`) correspond au polygone continu délimité par les centres des pixels de bord, et est donc inférieure au simple comptage des pixels à 1 (`np.sum`) ; et (ii) la liste des objets doit être triée par ordre croissant selon la coordonnée X de la boîte englobante (`bbox[0]`), avec la coordonnée Y (`bbox[1]`) comme critère de départage, les identifiants étant réattribués séquentiellement de 1 à N après le tri.

Comme dans l'exemple de la Section A.3, le texte de l'énoncé n'est pas non plus fixe ici : il est construit à l'intérieur même du bloc `[[def: ... ]]`, dans la variable Python `texto`, et joue le même double rôle déjà décrit — il alimente le PDF via `[[code:texto]]` et est exporté, déjà converti par `latex_to_text()`, dans `cases['description']` au sein de `moodle_cases`, cette dernière étant la version affichée à l'étudiant dans le VPL de Moodle. La différence est que, dans ce cas réel, c'est le **scénario visuel** (image bruitée, nombre et position des objets) qui change d'un étudiant à l'autre à chaque variante, tandis que la rédaction de `texto` reste fixe entre les variantes, puisque seules les données numériques (image et cas de test) sont tirées. L'idéal serait que la rédaction varie elle aussi à chaque génération, comme dans l'exemple précédent, où la valeur de `height` était interpolée directement dans le texte de la description. La structure réellement utilisée (avec les extraits de tirage du scénario omis par souci de concision) est :

```
[[code:texto]]

\vspace{2mm}\noindent\textbf{Exemple d'entrée :}\vspace{-2mm}\
\begin{verbatim}
[[code:caso0_inp]]
\end{verbatim}
\vspace{-2mm}\noindent\textbf{Exemple de sortie :}\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 # DOIT inclure "topic." dans MCTest

# ... génération de la scène, du bruit et des 10 cas de test (inplist/outlist) ...

# ÉNONCÉ COMPLET, AVEC EXPLICATION DE L'AIRE ET DU TRI
texto = (
    "Une image binaire corrompue par un bruit poivre et sel contient au moins deux objets
    géométriques isolés.\n\n"
    "Écrivez un programme qui :\n"
    "1. Lit deux entiers ..."
)

cases = {}
cases['skills']      = ["morphologie mathematique", "suppression de bruit", "mm.measure",
"tabulation"]
cases['description'] = [{"text": latex_to_text(texto)}]
cases['input']       = np.array(inplist).tolist()
cases['output']      = np.array(outlist).tolist()

moodle_cases = json.dumps(cases)
caso0_inp = cases['input'][0]
caso0_out = cases['output'][0]
```

```
]]
```

Ci-dessous, la photographie de la question effectivement générée et imprimée pour l'une des 110 variantes du Simulado 4, montrant l'énoncé, l'image d'entrée bruitée et l'exemple de sortie attendu (image filtrée et tableau des mesures) :

1. Uma imagem binária corrompida por ruído sal e pimenta contém no mínimo dois objetos geométricos isolados.

Escreva um programa que: 1. Leia dois inteiros na primeira linha representando a altura (H) e largura (W) da imagem. 2. Leia as H linhas seguintes da matriz binária (valores 0 e 1 separados por espaço) usando mm.readImg(H, W). 3. Aplique a filtragem morfológica para eliminar o ruído sal e pimenta. 4. Imprima a matriz limpa diretamente em valores 0 e 1 (0s e 1s separados por espaço). 5. Extraia as medições geométricas dos objetos usando mm.measure(imgLimpa). 6. Ordene a lista de objetos e imprima a tabela formatada das medidas.

Observação sobre a área: A área calculada pelo OpenCV (cv2.contourArea) mede o polígono contínuo delimitado pelos centros dos pixels de borda, resultando em um valor menor do que a simples contagem discreta de pixels 1 (np.sum).

Regra de Ordenação: A lista de objetos deve ser ordenada em ordem crescente pela posição X do retângulo delimitador (bbox[0]), utilizando a posição Y (bbox[1]) como critério de desempate caso dois objetos comecem na mesma coluna (medidas.sort(key=lambda m: (m['bbox'][0], m['bbox'][1]))). Em seguida, reatribua os IDs sequencialmente de 1 a N.

Exemplo de Entrada:

```
12 43
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 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 0 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Exemplo de Saída:

```
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

id	area	perimeter	cx	cy	x	y	w	h	circularity	solidity	vertices
1	20	18.0	6.5	2.0	4	0	6	5	0.78	1.000	4
2	20	18.0	26.0	2.5	24	0	5	6	0.78	1.000	4

Figure A.2: Question réelle générée par MCTest pour le Simulado 4 — sujet « Opérateurs morphologiques ».

A.5 Remarques finales

Cette annexe a résumé les étapes pour installer MCTest, créer des examens et construire des questions paramétriques avec intégration au VPL de Moodle. L'utilisation combinée de variables tirées dans la description (`[[code:...]]`) et de code Python intégré à la question elle-même (`[[def: ... ]]`) permet de générer des centaines de variantes d'examen à partir d'un seul modèle, chacune corrigée automatiquement et de manière cohérente — la variable `texto`, en particulier, garantissant que le même énoncé est affiché aussi bien dans le PDF imprimé que lors de la consultation de la question par l'étudiant dans Moodle. L'exemple réel du Simulado 4 (Section A.4) illustre comment ce mécanisme est utilisé en pratique pour générer des questions de programmation avec des scénarios visuels différents pour chaque étudiant. L'Annexe B détaille comment ces variantes sont publiées et corrigées dans Moodle, et l'Annexe C décrit comment restreindre l'accès aux examens à l'aide de SEB.

Annexe B

À propos de Moodle (VPL)

Cette annexe décrit la configuration d'une activité **VPL (*Virtual Programming Lab*)** dans Moodle, le *plugin* utilisé pour la soumission et la correction automatique des Exercices de Programmation (EPs), des simulacres d'examen bimensuels et des examens décrits dans ce livre. La version de Moodle utilisée était la **4.5**.

B.1 Configurer une activité VPL

La création d'un examen dans MCTest avec intégration VPL (voir Annexe A) génère deux fichiers qui doivent être renommés en `linker.json` et `students_variations.csv`. Ces fichiers, avec `morph.py`, doivent être ajoutés aux **fichiers d'exécution** de l'activité VPL, ainsi que tous les fichiers contenus dans `VPL_modification/V14-AI-feedback.zip`, disponible dans le dépôt de MCTest à l'adresse https://github.com/fzampirolli/mctest. Tous doivent être marqués avec l'option « ***Files to keep when running*** ».

En résumé, l'activité VPL connaît alors, pour chaque étudiant, quelle variante de la question a été tirée (via `students_variations.csv`) et quel est l'ensemble de cas de test correspondant (via `linker.json`). Cela permet une correction individualisée, même lorsque des dizaines d'étudiants résolvent la même activité avec des variantes différentes.

B.2 Publier les EPs comme activités VPL

Les Exercices de Programmation (EPs) présentés à la fin de chaque chapitre de ce livre peuvent aussi être publiés comme activités VPL, en suivant le même mécanisme. Le déroulement type en classe était :

1. Ouverture des deux *notebooks* Colab du chapitre (théorique et pratique), en mettant l'accent sur les simulateurs interactifs ;
2. Exécution des blocs de code, en modifiant les paramètres pour renforcer la compréhension des concepts ;
3. Lors des séances en laboratoire, soumission des réponses aux EPs directement dans l'activité VPL correspondante, avec un retour immédiat sur la réussite ou l'échec des cas de test.

Si l'activité VPL n'est pas générée par MCTest, les fichiers `linker.json` et `students_variations.csv` ne sont pas nécessaires. Cependant, pour utiliser le *feedback* par IA (voir Annexe D), il faut ajouter les fichiers contenus dans `VPL_modification/V14-EP0_VPL-TEMPLATE.zip`, disponible dans le même dépôt (https://github.com/fzampirolli/mctest), en suivant la procédure décrite à la Section B.1.

> **Réutilisation des cas de test**
>
> Les fichiers `.cases` utilisés lors de la validation locale (classe `TestSuite`, décrite dans le corps de ce livre) sont compatibles avec le format attendu par VPL, ce qui permet de réutiliser le même ensemble de tests aussi bien lors du développement de l'EP que lors de la correction automatisée dans Moodle.

Lors des simulacres d'examen bimensuels — appliqués avec SEB (voir Annexe C) et donnant droit à un bonus de 5 % sur la note finale —, des activités VPL ont été utilisées avec des EPs semblables à ceux des listes régulières, mais corrigées avec un accès à internet restreint, renforçant le caractère évaluatif de l'activité.

B.3 Remarques finales

Cette annexe a décrit la configuration des activités VPL dans Moodle pour la correction automatique des EPs, des simulacres d'examen et des examens paramétrés. La combinaison de MCTest (génération des variantes et des cas de test), de VPL (soumission et correction) et de SEB (restriction d'accès) forme le tripode sur lequel repose l'évaluation automatisée et individualisée utilisée tout au long du cours.

Annexe C

Utilisation du SEB lors des simulacres d'examen bimensuels et des évaluations

Outre les deux examens à questions paramétrées (voir Annexe A), le ***Safe Exam Browser*** **(SEB)** a également été utilisé pour restreindre l'accès lors des **simulacres d'examen bimensuels** — activités VPL (*Virtual Programming Lab*) avec des Exercices de Programmation (EPs) semblables à ceux des listes régulières, donnant droit à un bonus pouvant aller jusqu'à 5 % de la note finale —, ainsi que lors des autres évaluations pratiques.

Le SEB garantit un environnement d'évaluation sécurisé en bloquant l'accès aux ressources externes non autorisées sur les machines du laboratoire. Pour la mise en œuvre adoptée dans ce travail, le système d'exploitation **Windows 10** et la version **3.7.1.704** du SEB, disponible sur https://safeexambrowser.org/, ont été utilisés ; ils doivent être préalablement installés sur tous les postes du laboratoire.

La procédure de configuration du fichier d'environnement du SEB et de son association à l'activité VPL dans Moodle est décrite ci-dessous.

C.1 Configurer l'application dans *SEB Tool*

Pour générer le fichier de configuration `.seb` qui sera distribué aux étudiants, ouvrez l'utilitaire *SEB Tool* et suivez les étapes ci-dessous.

1. ***General***
   - Dans le champ ***Start URL***, saisissez l'URL directe de l'activité VPL dans Moodle.
   - *Remarque :* s'il est nécessaire de naviguer entre plusieurs activités au sein du même cours, saisissez l'URL principale du cours et, dans l'onglet ***Browser***, cochez l'option ***Allow navigation back/forward in exam***.
2. ***Network***
   - Incluez les URL autorisées pour la navigation. Pour restreindre l'accès à l'activité souhaitée, utilisez une expression de filtre du type `/mod/vpl/*?id=4624*`, où le numéro final correspond à l'identifiant de l'activité VPL dans Moodle. Dans le Table C.1, la dernière ligne contient l'URL de la page principale du cours (identifiant `475`), mais elle peut aussi être remplacée par l'URL d'une section contenant plusieurs activités. Ce réglage est facultatif et utile lorsqu'il y a plus d'une activité VPL. Dans ce cas, la première ligne du tableau doit être répétée pour chaque activité évaluative devant rester accessible pendant l'*examen*.
   - **Réglages généraux de l'onglet *Filter*** :
     ☒ *Activate URL filtering*
     ☐ *Filter also embedded content*

Tableau C.1: Expressions de filtrage d'URL utilisées dans la configuration du 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*

Active	*Regex*	*Expression*	*Action*
[x]	[ ]	`https://moodle.ufabc.edu.br/course/switchrole.php*`	*Allow*
[x]	[ ]	`https://mctest.ufabc.edu.br/compare`	*Allow*
[x]	[ ]	`https://moodle.ufabc.edu.br/enrol/index.php?id=475`	*Allow*

3. ***Security***
   - Si le laboratoire utilise un projecteur ou un second écran, réglez le champ ***Maximum allowed number of connected displays*** sur une valeur supérieure à 1, pour éviter que le SEB ne bloque l'affichage.
4. ***Config File***
   - Cliquez sur ***Save Settings As...*** et enregistrez le fichier avec l'extension `.seb`.
5. ***Exam***
   - Cochez l'option ***Use Browser Exam Key and Configuration Key***.
   - Copiez le code généré dans le champ ***Browser Exam Key***. *Important :* copiez cette clé seulement **après** avoir enregistré le fichier à l'étape précédente.

C.2 Associer la clé SEB à l'activité VPL dans Moodle

Après avoir généré le fichier et obtenu la *Browser Exam Key*, accédez à Moodle pour appliquer les restrictions.

1. Sur la page de l'activité VPL, cliquez sur l'icône d'engrenage (**Paramètres**).
2. Accédez à ***Configuration*** → ***Submission restrictions*** et cliquez sur ***Show more***.
3. Réglez l'option ***SEB browser required*** sur ***Yes***.
4. Collez la clé copiée dans le champ ***SEB exam key(s)***.

C.3 Restriction par adresse IP (facultatif)

Pour garantir que les soumissions proviennent exclusivement des ordinateurs du laboratoire, il est possible de combiner le SEB avec le filtre réseau de Moodle, en renseignant le champ ***Allowed submission from net***.

- **Plages consécutives :** utilisez un tiret pour définir l'intervalle. Exemple : `111.11.11.1-62` (autorise les IP de `111.11.11.1` à `111.11.11.62`).
- **IP non consécutives :** indiquez les adresses individuelles séparées par des virgules.

C.4 Remarques finales

L'intégration du SEB avec le VPL dans Moodle crée un déroulement simple pour l'étudiant : il suffit de double-cliquer sur le fichier `.seb` fourni par l'enseignant pour que le navigateur sécurisé s'ouvre directement sur l'activité configurée.

La combinaison de la *Browser Exam Key* avec la restriction par plage d'adresses IP des machines du laboratoire fournit une solution robuste pour les évaluations de programmation *en ligne*, rendant plus difficiles les accès indus et contribuant à l'intégrité du processus d'évaluation.

Annexe D

Génération de matériel pédagogique avec *Gemini Notebook*

Cette annexe décrit l'utilisation du ***Gemini Notebook*** (***NotebookLM***) comme outil d'appui à la préparation du matériel pédagogique du cours, complétant l'usage de l'IA pour la révision des textes et des codes, mentionné dans la préface de ce livre.

D.1 Vidéos conceptuelles et de résolution des EPs

Pour chaque chapitre, un projet a été créé dans *Gemini Notebook* à partir du PDF complet du livre et du fichier `morph.py` comme sources. À partir de ces sources, *Gemini Notebook* a généré automatiquement une courte vidéo (7 minutes en moyenne), utilisée en ouverture des cours. Les chapitres alternaient deux types de vidéo :

- **Vidéo conceptuelle**, présentant les fondements théoriques du chapitre, généralement diffusée lors du premier cours consacré au sujet ;
- **Vidéo de résolution des EPs**, présentant une stratégie de résolution des Exercices de Programmation, diffusée lors du cours suivant, lorsque la plupart des EPs étaient résolus collectivement avec la classe.

> **ℹ Rôle de la vidéo en classe**
>
> La vidéo générée par *Gemini Notebook* ne remplace pas l'explication de l'enseignant. Elle sert de brève introduction et de motivation au sujet, contextualisant l'étudiant avant la présentation des *slides* et l'ouverture des *notebooks* Colab.

D.2 *Slides* d'appui

En complément des vidéos, *Gemini Notebook* a également généré, à partir des mêmes sources (le PDF du livre et le fichier `morph.py`), un ensemble d'environ 12 *slides* par chapitre, couvrant à la fois les concepts théoriques et la partie pratique. La génération utilisait un *prompt* spécifique à chaque chapitre, délimitant la portée du contenu à synthétiser et évitant à la fois d'anticiper des sujets de chapitres ultérieurs et de reproduire de longs passages du livre sans adaptation pédagogique.

Après la présentation des *slides*, le cours se poursuivait par l'ouverture des deux *notebooks* Colab du chapitre (théorique et pratique), en mettant l'accent sur les simulateurs interactifs et l'exécution des blocs de code avec modification des paramètres.

D.3 Matériel principal et remarques finales

La génération de vidéos et de *slides* avec *Gemini Notebook*, à partir du livre lui-même et de la bibliothèque `morph.py`, a permis de standardiser l'ouverture des cours et de réduire le temps de préparation du matériel pédagogique, sans se passer pour autant de la curation de l'enseignant dans l'élaboration des *prompts* et la conduite des activités en classe.

Les vidéos et *slides* produits par *Gemini Notebook* ont un caractère **motivationnel** et servent de point de départ pour discuter des concepts présentés dans chaque chapitre. Comme tout contenu généré par IA, ils peuvent contenir des imprécisions ou des erreurs occasionnelles. Dans certains cas, ces imprécisions sont exploitées intentionnellement pendant le cours pour vérifier si les étudiants suivent la présentation de manière critique et parviennent à identifier des incohérences conceptuelles.

Le contenu officiel du cours est cependant le matériel produit selon la méthode décrite dans la préface de ce livre, disponible en trois formats complémentaires : **HTML**, pour la navigation avec des simulateurs interactifs ; **PDF**, pour la lecture et l'impression ; et *notebooks* **Jupyter** (`.ipynb`), pour l'exécution et l'expérimentation des codes. Tout le matériel généré par *Gemini Notebook* doit être compris comme une ressource complémentaire à ce contenu principal.

Comme pour les autres usages de l'IA générative décrits dans ce livre, la responsabilité de la qualité technique, de la justesse conceptuelle et de l'adéquation pédagogique du matériel demeure celle de l'enseignant.

Annexe E

Utilisation de l'IA dans l'évaluation des étudiants : le projet vpl-ai-feedback

Cette annexe décrit **vpl-ai-feedback**, un système *open source* développé pour appuyer la correction des soumissions de code envoyées au VPL (Moodle) grâce à un *feedback* généré par Intelligence Artificielle (IA). Contrairement à un correcteur automatique traditionnel — qui se contente de comparer les sorties aux cas de test —, **vpl-ai-feedback** envoie le code de l'étudiant, accompagné de la grille de correction de la question, à un modèle de langage (LLM), qui renvoie une analyse qualitative par critère, dans le style d'un ***feedback* socratique** : l'étudiant reçoit des commentaires indiquant ce qui a été fait correctement, ce qui peut être amélioré et pourquoi, aussi bien dans les activités **formatives** (exercices d'entraînement) que dans les activités **évaluatives** (simulacres d'examen) du VPL.

Ce mécanisme de *feedback* socratique dans les activités VPL a été proposé par Zampirolli et al. (2026). L'étude décrit l'intégration de l'IA au processus d'évaluation d'exercices de programmation paramétrés dans VPL/Moodle, à l'aide d'un ensemble de sept LLM utilisés pour analyser le code des étudiants et générer un *feedback* automatisé à chaque demande de correction, avec des résultats préliminaires indiquant une perception positive des étudiants quant à la génération d'idées, à la clarté des explications et à l'autonomie d'apprentissage.

Le reste de cette annexe détaille la mise en œuvre pratique de ces idées dans le projet **vpl-ai-feedback** — dont le code est disponible à l'adresse https://github.com/fzampirolli/vpl-ai-feedback — et décrit spécifiquement son utilisation dans les trois classes de PDI-VC entre mai et août 2026.

> ⚠ Attention : l'IA ne remplace pas le jugement de l'enseignant
>
> L'utilisation de l'IA pour **évaluer** les étudiants **n'est pas recommandée comme source unique de note**, car les modèles de langage peuvent **halluciner** — c'est-à-dire produire des évaluations, des critères ou des justifications plausibles mais incorrects, même face à un code correct ou incorrect. Un LLM peut attribuer une note élevée à un code comportant une erreur subtile, ou une note faible à un code correct mais écrit de façon inhabituelle, simplement parce que l'explication générée « paraît » cohérente. **L'IA doit être utilisée uniquement comme un complément à l'évaluation**, jamais comme un substitut à l'enseignant. L'Annexe A et l'Annexe B décrivent les mécanismes de correction objective (cas de test) qui restent la base de la note officielle ; cette annexe traite exclusivement de l'utilisation de l'IA comme couche supplémentaire de *feedback* qualitatif.

E.1 Contexte d'utilisation

vpl-ai-feedback a été utilisé dans trois classes de la discipline **PDI-VC**, enseignée entre mai et août 2026, totalisant environ une centaine d'étudiants. Le *feedback* par IA a été employé de trois façons complémentaires :

1. ***Feedback* continu dans les activités formatives** — exercices d'entraînement soumis au VPL tout au long du cours, où l'étudiant pouvait soumettre à nouveau son code autant de fois qu'il le souhaitait.

À chaque soumission, il recevait un *feedback* qualitatif généré par IA, en plus de la correction objective effectuée par le VPL, décrite dans Zampirolli et al. (2026).

2. ***Feedback* complémentaire lors des simulacres d'examen (activités évaluatives bonus)** — quatre simulacres réalisés au cours du trimestre, prévus dans le plan de cours comme activités bonus, chacun valant jusqu'à 5 % de la note finale. Appliqués environ 30 minutes avant la fin des séances pratiques bimensuelles en laboratoire, les simulacres utilisaient **vpl-ai-feedback** pour générer une grille d'évaluation individuelle, envoyée par e-mail avec un résumé comparatif entre la note attribuée par Moodle/VPL et l'évaluation produite par l'IA.

3. ***Feedback* complémentaire lors des Examens 1 et 2 (évaluations officielles)** — les deux examens formels du trimestre, chacun valant une part plus importante de la note finale du cours, ont également commencé à utiliser **vpl-ai-feedback** après leur passation. Contrairement aux simulacres (bonus), la note officielle est ici déjà définie par la correction objective du VPL et/ou l'évaluation manuelle de l'enseignant avant la publication des résultats ; la grille générée par l'IA n'est envoyée à l'étudiant qu'ensuite, comme support pour la révision de sa propre performance — renforçant, y compris dans les évaluations à plus fort coefficient, la séparation entre *note officielle* et *feedback* complémentaire détaillée à la Section D.7.

Une enquête d'opinion, à laquelle 102 étudiants ont répondu avant l'adoption du système, a indiqué une forte adhésion à la proposition. Seuls 2 étudiants ont attribué la note 1 (rejet) et 7 ont attribué la note 2 quant à l'intérêt de recevoir un *feedback* automatique de code par IA. 19 autres se sont déclarés indifférents (note 3), tandis que la majorité — 38 et 37 étudiants — a attribué les notes 4 et 5 (forte approbation), respectivement, totalisant les 102 répondants. Ce résultat a motivé l'adoption du système comme complément au processus formatif, et non comme substitut à la correction humaine.

! La note officielle n'a jamais été la note de l'IA

Il est essentiel de souligner que, comme défini dans le plan de cours, **la note utilisée aussi bien pour le calcul du bonus de chaque simulacre que pour la note officielle des Examens 1 et 2 a toujours été la note attribuée par le VPL** (basée sur les cas de test) et/ou par l'évaluation manuelle de l'enseignant — et **non** la note suggérée par l'IA. La grille générée par **vpl-ai-feedback** a été envoyée à l'étudiant uniquement comme support d'apprentissage — jamais comme critère d'attribution de note, que ce soit dans les activités bonus ou dans les évaluations officielles. Cette séparation entre *note officielle* (VPL/enseignant) et *feedback complémentaire* (IA) est le principe central de cette annexe et a été reprise à la Section D.8.

E.2 Architecture du projet

vpl-ai-feedback est un *script* Python asynchrone organisé de la façon suivante :

```
.
   config.yaml              ← configuration (identifiants, fournisseur, poids) - JAMAIS
versionné
   config.yaml.example      ← modèle public de configuration
   main.py                  ← script principal (async), orchestre tout le processus
   run.sh                   ← wrapper d'exécution (bash run.sh config.yaml)
   gerar_relatorio.py       ← convertit le fichier consolidé *_ALL.txt en CSV
   enviar_email.py          ← envoie les grilles par e-mail à chaque étudiant
   providers/                ← clients des API de LLM
       __init__.py          ← fabrique de clients (Factory)
       base.py               ← logique de retry, backoff et repli entre modèles
       groq.py
       deepseek.py
       gemini.py
   core/                      ← logique métier
```

```
      grader.py              ← identifie les fichiers par question, construit le prompt,
appelle le LLM
      utils.py
   <dossier_de_la_classe>/    ← soumissions téléchargées depuis Moodle VPL
       Nom étudiant - login/
           <timestamp>/                 ← dernière soumission de l'étudiant
               Q1.py                    ← modèle pour les examens à plusieurs questions
               Q2.py
               rubrica.txt              ← généré par le LLM (uniquement si l'évaluation est
    valide)
           <timestamp>.ceg/
               execution.txt            ← note/objective attribuée par le VPL
       ...
```

Le système accepte trois fournisseurs de LLM — **Groq**, **DeepSeek** et **Gemini** — configurables dans `config.yaml` via la clé `llm.provider`. Chaque fournisseur peut avoir **plusieurs modèles** enregistrés ; à chaque appel, la liste est **mélangée**, répartissant la charge entre les étudiants traités en parallèle et réduisant le risque d'erreurs de limite de requêtes (HTTP 429). Lorsqu'un modèle échoue, le système réessaie jusqu'à trois fois avant de passer au suivant dans la liste, en respectant le délai d'attente suggéré par le fournisseur. Les erreurs d'authentification (401) ou de solde insuffisant (402) interrompent immédiatement la tentative auprès de ce fournisseur.

Un aspect important du projet concerne la confidentialité des données. **Le nom, l'e-mail, le login, le numéro d'étudiant et le numéro d'identité nationale de l'étudiant ne sont jamais envoyés à l'API du LLM**. Ne sont transmis pour traitement que : le nom du fichier contenant le code source ; le code lui-même (sans les commentaires) ; le *prompt* de la grille — qui ne contient pas de données personnelles — ; et, lorsqu'ils sont disponibles dans le fichier `execution.txt` généré par le VPL, l'énoncé officiel de la question tirée pour cet étudiant et le résultat brut des cas de test réellement exécutés (entrée, sortie produite par le code et sortie attendue). Ces deux derniers éléments, bien qu'extraits d'un fichier spécifique à chaque étudiant, ne contiennent pas non plus d'identification personnelle — seulement le texte de la question et les valeurs d'entrée/sortie des tests automatiques —, et sont utilisés comme preuve objective pour réduire le risque que l'IA identifie incorrectement le type de question ou évalue la logique du code par simple lecture statique (Section D.3).

Il n'en demeure pas moins important de reconnaître que les trois fournisseurs actuellement pris en charge — Groq, DeepSeek et Gemini — sont des services commerciaux exploités par des entreprises étrangères, dont les modèles s'exécutent sur une infrastructure située hors du pays et sont soumis à des politiques d'utilisation, de disponibilité et de coût qui échappent au contrôle de l'établissement d'enseignement. Cette dépendance à des fournisseurs externes comporte des risques qui vont au-delà de la simple correction ponctuelle de code : changements de prix, abandon de modèles, indisponibilité temporaire du service, modifications unilatérales des conditions d'utilisation et, dans les cas les plus sensibles, restrictions géopolitiques d'accès peuvent compromettre la continuité d'un système d'évaluation devenu structurellement dépendant de ces API. C'est pourquoi il est stratégique que des établissements d'enseignement supérieur comme l'UFABC investissent dans le développement et l'hébergement de **leurs propres fournisseurs d'IA** — par exemple, des modèles ouverts (*open-weight*) exécutés sur une infrastructure locale ou un cloud souverain —, réduisant la dépendance aux services externes, en particulier d'autres pays, et élargissant le contrôle institutionnel sur les coûts, la disponibilité, la confidentialité des données des étudiants et la continuité pédagogique à long terme. La conception de **vpl-ai-feedback** favorise déjà cette voie : l'architecture de `providers/` a été construite comme une fabrique (*factory*) extensible, dans laquelle un nouveau fournisseur — y compris un modèle hébergé localement par l'établissement lui-même, avec une interface compatible — peut être ajouté avec un faible effort d'intégration.

E.3 Le *prompt* universel et la grille en deux étapes

Chaque type d'examen est décrit dans un fichier de *prompt* (par exemple, `Simulado4.txt`), référencé dans `grading.prompt_file` du `config.yaml`. Ce fichier indique au LLM d'effectuer l'évaluation en **deux étapes**

:

1. **Identification de la question** — le LLM détermine le type spécifique d'opération tiré pour cet étudiant (utile lorsqu'il existe des questions paramétrées tirées et mélangées par étudiant, comme décrit à l'Annexe A) ;
2. **Application de la grille correspondante** — le LLM évalue le code selon des critères notés, chacun avec un barème maximal de points, renvoyant à la fin une note au format `NOTE FINALE : X/POIDS`.

Dans les examens paramétrés, le fichier de *prompt* contient généralement **plusieurs grilles parallèles**, une pour chaque type de question possible, délimitées par des marqueurs tels que `[START_RUBRICA_TIPO_A]` / `[END_RUBRICA_TIPO_A]`. Le LLM identifie d'abord quel type a été tiré pour cet étudiant et applique **uniquement** la grille correspondante, en ignorant les autres — ce qui évite que des critères d'un autre type ne contaminent la note.

Lorsque l'examen comporte plusieurs questions par activité VPL, les fichiers de code doivent suivre le modèle `Q1.*`, `Q2.*`, etc. — un fichier par question, avec une extension libre selon le langage choisi par l'étudiant —, et chaque question a un poids configurable individuellement dans `grading.weights` du `config.yaml`. Lorsque l'examen ne comporte qu'une seule question et n'a pas été généré par MCTest, le système accepte n'importe quel nom de fichier avec une extension prise en charge, à condition qu'il soit le seul fichier de code présent dans le dossier de soumission.

Dans le cas du Simulado 4 (sujet *Opérateurs morphologiques*), par exemple, l'un des types possibles définissait trois critères, pour un poids total de 100 points pour cette question :

Critère	Description	Note maximale
1 — Entrée des données	Lecture de `H`, `W` et de la matrice binaire (`mm.readImg`)	25 pts
2 — Traitement morphologique	Ouverture + Fermeture (`mm.sebox()`), `mm.measure`, tri par `bbox[0]`/`bbox[1]` et réattribution des identifiants	50 pts
3 — Sortie et mise en forme	Affichage de l'image nettoyée (`mm.drawImg`) et tableau des mesures formaté	25 pts

Le *prompt* exige également un **format de réponse obligatoire** (largeur de ligne, ordre des sections, marqueur `NOTE FINALE :`), ce qui rend l'extraction automatique de la note et la constitution du rapport consolidé plus fiables — bien que, comme discuté à la Section D.8, cela n'élimine pas le risque d'erreur sémantique dans l'évaluation du contenu.

E.4 Deux couches supplémentaires de preuves

Pour réduire le risque que le LLM hallucine le type de question ou la justesse du code — le risque central discuté à la Section D.7 —, **vpl-ai-feedback** a commencé à fournir à l'IA deux sources de preuves objectives, extraites automatiquement du fichier `execution.txt` généré par le VPL, en plus du code source :

- **Énoncé officiel de la question.** Comme les questions sont généralement tirées et mélangées par étudiant dans MCTest (Annexe A), le `execution.txt` de chaque étudiant contient déjà, dans le champ « Résumé de la question », le texte exact de la question qui lui a été attribuée. Le système extrait automatiquement cet extrait et l'envoie au LLM comme preuve **primaire** pour identifier le type/la variante de la question — plus fiable que de se fier uniquement à une lecture statique du code, en particulier pour les soumissions incomplètes ou ambiguës entre deux types proches (par exemple, une érosion « plate » *versus* « pondérée »). Lorsque le code soumis diverge de ce que demande l'énoncé, le système n'ignore pas l'énoncé et ne « corrige » pas silencieusement la lecture : il conserve l'identification

fondée sur l'énoncé, réduit le niveau de confiance de l'évaluation à « Faible » et explicite la divergence dans le *feedback* envoyé à l'étudiant.

- **Résultat réel de l'exécution sur le VPL.** Lorsqu'ils sont disponibles, le système envoie aussi au LLM les cas de test effectivement exécutés sur ce code — entrée, sortie produite et sortie attendue — comme preuve objective de correction, à utiliser pour confirmer ou infirmer la lecture de la logique avant de noter les critères de traitement et de sortie, plutôt que de laisser l'IA se fier uniquement à une simulation mentale du code (ce qui est particulièrement défaillant dans les boucles avec `min/max`, les décalages d'indices ou le calcul du seuil d'Otsu, où des erreurs subtiles passent inaperçues lors d'une lecture statique).

De plus, le LLM est instruit de déclarer explicitement son **niveau de confiance** (Élevé, Moyen ou Faible) dans l'identification de chaque type de question, avec justification lorsque la confiance n'est pas Élevée. Cette valeur est extraite automatiquement et alimente une colonne `Revisar_Manualmente` (à réviser manuellement) dans le rapport CSV consolidé (Section D.4), permettant à l'enseignant de prioriser exactement les cas où l'IA elle-même reconnaît une plus grande incertitude — plutôt que de traiter toutes les évaluations d'une classe comme également fiables.

E.5 Déroulement de l'exécution

L'exécution type, effectuée via `bash run.sh config.yaml`, suit les étapes suivantes :

1. Charge `config.yaml` et localise le dossier des soumissions (`paths.student_base_dir`) ;
2. Pour chaque étudiant, identifie la **dernière soumission** (le dossier au *timestamp* le plus récent) ;
3. Extrait la note objective de Moodle à partir de `*.ceg/execution.txt` ;
4. Pour chaque question de l'examen, localise le fichier de code (`Q1.*`, `Q2.*`, ou le seul fichier, dans le cas d'un examen non généré par MCTest), construit le *prompt* (code + grille) et l'envoie à un modèle tiré de la liste configurée ;
5. Traite tous les étudiants **en parallèle**, avec un contrôle de la concurrence ;
6. Génère, pour chaque étudiant, le fichier `rubrica.txt` — **uniquement si l'IA renvoie au moins une évaluation valide** pour les questions comportant du code ; sinon, le fichier n'est pas enregistré, ce qui force une nouvelle tentative à l'exécution suivante ;
7. Consolide tous les fichiers `rubrica.txt` en un seul `*_ALL.txt` et génère un rapport `*_relatorio.csv`, comparant la note Moodle, la note de l'IA et l'écart entre les deux, par question et au total.

Situation	`rubrica.txt` est-il enregistré ?
Toutes les questions avec code évaluées avec succès	✅ Oui
L'IA a échoué sur au moins une question avec code	❌ Non — retraité à l'exécution suivante
Étudiant n'ayant soumis aucun fichier de code	❌ Non
Type de question identifié comme INCONNU	❌ Non

E.6 Exemple illustratif : grille pour un examen à trois questions

> ℹ À propos de l'origine de cet exemple
>
> L'extrait suivant **ne reproduit le code d'aucun étudiant en particulier**. Il s'agit d'un exemple reconstruit par l'auteur, reproduisant un *schéma* d'erreur observé de façon répétée dans les classes de PDI-VC — la confusion entre une fonction morphologique 2D (`mm.ero/mm.ero0`) et une opération 1D sur des signaux — sans citer ni identifier aucune soumission réelle. Ce choix évite tout risque d'atteinte au droit d'auteur de l'étudiant sur son propre code source, même anonymisé, la paternité de l'œuvre restant celle de l'auteur original même lorsque le nom et le *login* sont retirés.

L'extrait suivant illustre la sortie générée par **vpl-ai-feedback** pour un étudiant sur un examen à **trois**

questions (Q1, Q2, Q3), avec le modèle `deepseek-chat`. Le résumé comparatif apparaît en haut du fichier, suivi de l'énoncé officiel de chaque question, du code soumis et de l'évaluation par critère.

```
RÉSUMÉ - IA (deepseek-chat)  x  MOODLE

Poids 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

Différence (IA - Moodle) : -23.7 pts

Q1 : type identifié = B  | confiance = Faible (le code ne
  correspond pas à l'énoncé de la question)   À RÉVISER
Q2 : type identifié = A  | confiance = Élevée (l'énoncé officiel
  confirme le type et la variante)
Q3 : type identifié = C1 | confiance = Élevée (l'énoncé officiel
  confirme une érosion 2D avec OpenCV intégrée)
```

Q1 — le code ne correspond pas à l'énoncé (confiance faible) : l'énoncé officiel de la question, extrait de `execution.txt`, demandait une érosion morphologique **1D** sur un signal (lecture de N, M, du signal et de l'élément structurant B, suivie du minimum des voisins actifs). Le code soumis, cependant, ne lit que deux entiers et appelle `mm.ero0` — une fonction d'érosion **2D** pour images, sans rapport avec l'opération demandée.

```
Type identifié : TYPE B (Érosion 1D - plate)
Confiance : Faible - le code ne correspond pas à l'énoncé de la
question (utilise mm.ero0, fonction d'érosion 2D, au lieu d'implémenter
une érosion 1D plate sur un signal)

Critère 1 - [3.33/10.00 pts] :
- Le code lit deux entiers sur une seule ligne (qui correspondraient à
  N et M), mais ne lit pas le signal ni l'élément structurant B comme
  des vecteurs 1D. Il les traite plutôt tous les deux comme des images
  via mm.readImg, ce qui ne correspond pas à la lecture attendue d'un
  signal 1D et de B sur des lignes séparées.

Critère 2 - [0.00/16.67 pts] :
- L'opération appliquée est mm.ero0(img, element), une érosion 2D pour
  images, et non l'érosion 1D de signaux demandée dans l'énoncé. La
  logique du minimum des voisins actifs avec rognage aux bords n'est
  pas implémentée.

Critère 3 - [0.00/6.67 pts] :
- La sortie est une image 2D via mm.drawImg, et non le signal 1D
  résultant sur une seule ligne avec des valeurs entières séparées par
  des espaces, comme requis.

NOTE FINALE : 3.33/33.33

Commentaires pour l'étudiant : le code présenté ne correspond pas à ce
que demande l'énoncé. La question tirée demande une érosion
morphologique 1D sur un signal unidimensionnel ; le code utilise une
fonction d'érosion 2D pour images. Relisez attentivement l'énoncé tiré
```

```
avant de choisir l'implémentation.
```

Q2 — une erreur de syntaxe empêche l'exécution (confiance élevée, note faible due à un véritable échec) : ici, l'IA a correctement identifié le type (seuillage avec Otsu), en s'appuyant sur l'énoncé officiel, mais le code contenait une erreur d'indentation qui empêchait l'exécution — mise en évidence à la fois par la lecture statique et par l'absence de sortie valide dans les tests du VPL.

```
Type identifié : TYPE A (Seuillage - T manuel + Otsu)
Confiance : Élevée - l'énoncé officiel confirme le type et la variante

Critère 1 - [6.67/10.00 pts] :
- Lit correctement les trois paramètres L, C et T sur des lignes
  séparées ainsi que l'image d'entrée. Cependant, la définition d'une
  fonction auxiliaire de seuillage est mal indentée dans la portée
  globale, provoquant une erreur d'indentation qui empêche l'exécution
  complète du programme.

Critère 2 - [0.00/16.67 pts] :
- La binarisation avec un seuil fixe T est correctement structurée,
  mais le calcul du seuil d'Otsu dépend de la fonction auxiliaire mal
  formée, qui référence une variable hors de portée. L'erreur
  d'indentation empêche tout traitement.

Critère 3 - [0.00/6.67 pts] :
- En raison de l'erreur d'indentation, aucune sortie n'est produite.

NOTE FINALE : 6.67/33.33

Commentaires pour l'étudiant : la structure générale est correcte
(lecture de L, C, T et de l'image), mais il y a une erreur
d'indentation qui empêche l'exécution. Révisez la syntaxe Python avant
de soumettre et testez le code localement avant l'envoi au VPL.
```

Q3 — divergence de fonction, mais équivalence fonctionnelle confirmée par une preuve réelle (confiance élevée, note juste) : l'énoncé demandait explicitement l'utilisation de `mm.ero` (version intégrée à OpenCV), mais le code utilise `mm.ero0` (version plate implémentée manuellement). Comme `execution.txt` montrait les 10 cas de test du VPL réussis intégralement, l'IA n'a pas pénalisé la divergence de fonction — elle a reconnu l'équivalence fonctionnelle démontrée par les tests réels, tout en consignant l'observation dans le *feedback*.

```
Type identifié : TYPE C1 (Érosion 2D - plate sans poids)
Confiance : Élevée - l'énoncé officiel confirme une érosion 2D avec
OpenCV intégrée (mm.ero), et le code implémente une érosion plate via
mm.ero0

Critère 1 - [10.00/10.00 pts] :
- Lit correctement la hauteur, la largeur, la matrice de l'image ainsi
  que les dimensions et valeurs de l'élément structurant B.

Critère 2 - [16.67/16.67 pts] :
- L'énoncé demandait l'utilisation de mm.ero (OpenCV intégrée), mais le
  code utilise mm.ero0 (plate, sans poids). Malgré la divergence de
  fonction, l'opération a été correctement implémentée avec l'élément
  structurant lu : les 10 tests exécutés sur le VPL ont réussi
  intégralement (10/10), confirmant l'équivalence fonctionnelle pour
  les cas testés.

Critère 3 - [6.67/6.67 pts] :
- Affiche l'image résultante dans le format correct ; les tests ont
  confirmé la sortie correcte.
```

```
NOTE FINALE : 33.34/33.33

Commentaires pour l'étudiant : le code est correct et fonctionnel, avec
les 10 tests réussis. Remarque : l'énoncé demandait explicitement
mm.ero (version OpenCV), mais vous avez utilisé mm.ero0 (version
plate). Bien que les résultats aient été équivalents dans les tests,
suivez exactement la fonction demandée dans l'énoncé lors des prochains
examens.
```

Notez que, dans ce cas composite, l'IA a attribué **23,7 points de moins** que la note objective du VPL, principalement à cause de Q1. L'écart agrégé constituerait déjà à lui seul un signal justifiant une investigation (Section D.7), mais le système fournit lui-même à l'enseignant la cause probable de chaque question : la colonne `Revisar_Manualmente` du CSV est marquée « OUI » chaque fois qu'au moins une question de l'étudiant reçoit une confiance faible, avec le motif spécifique consigné dans la colonne de confiance correspondante — réduisant le temps que l'enseignant doit consacrer à repérer, parmi une centaine d'étudiants, les rares cas qui méritent réellement une attention manuelle avant l'attribution de la note officielle. Le cas de Q3, quant à lui, illustre la précaution inverse : la divergence entre l'énoncé et le code **n'a pas** entraîné de pénalisation indue, car la preuve réelle d'exécution a confirmé l'équivalence fonctionnelle — exactement le comportement que décrit la Section D.3 pour ces deux couches de preuves.

E.7 Envoi du *feedback* par e-mail

Après la génération des grilles, le *script* `enviar_email.py` envoie à chaque étudiant un e-mail avec `rubrica.txt` en pièce jointe. Le corps de l'e-mail est défini dans `config.yaml`, sous `templates.corpo`, et est interpolé avec `{nome_pasta}` et `{login}` :

```yaml
email:
  smtp_server: smtp.ufabc.edu.br
  smtp_port: 587
  from_address: professor@ufabc.edu.br
  password: "VOTRE_MOT_DE_PASSE_ICI"     # ne jamais versionner de vrais identifiants
  use_tls: true

templates:
  assunto: "Grilles et corrections générées par LLM - Simulacre - {login}@aluno.ufabc.edu.br"
  corpo: |
    Cher/Chère {nome_pasta},
    ...
```

! Sécurité des identifiants

Le `config.yaml` contient de véritables secrets (mot de passe SMTP, clés d'API). Il doit figurer dans le `.gitignore` du projet et **ne jamais** être publié, partagé ou versionné — y compris en pièce jointe d'e-mail, en capture d'écran ou dans des dépôts publics.

Un exemple réel d'e-mail envoyé à un étudiant (données **anonymisées**, avec nom et login remplacés par un cas fictif) est reproduit ci-dessous, pour illustrer le ton pédagogique et les réserves explicitées à l'étudiant :

! Exemple de message individuel (anonymisé)

Cher/Chère [Nom de l'étudiant] - [login],
Votre note du Simulado 4 est disponible sur Moodle.
Je vous transmets ci-dessous les compétences évaluées ainsi qu'une correction détaillée de votre code, générée automatiquement par Intelligence Artificielle (IA).

La pièce jointe « rubrica.txt » contient le retour complet de l'IA (je recommande de la télécharger et de l'ouvrir avec un éditeur de texte ou le bloc-notes).
Je souligne que cette correction générée par IA PEUT contenir des imprécisions ou des erreurs, mais elle constitue un excellent support pour votre processus d'apprentissage dans le cours.
Une suggestion pédagogique consiste à soumettre votre code (contenu dans ce fichier), avec la GRILLE ci-dessous, à d'autres modèles de LLM pour comparaison. Cela peut aider à identifier d'éventuelles divergences, en plus d'offrir différentes perspectives sur votre code et sur les critères d'évaluation.
Si vous avez des questions ou remarquez une incohérence flagrante dans la correction présentée sur Moodle, je reste à votre disposition pour des éclaircissements.
Cordialement,
Prof. Francisco Zampirolli
PS. : Explication du processus : la dernière version de votre code enregistrée sur Moodle a été jointe au prompt et envoyée à l'un des LLM choisis de façon intégrée par notre système d'évaluation (modèles = ("deepseek-chat")). Le processus est répété jusqu'à l'obtention d'une réponse avec une note valide (entre 0 et 100).

Trois éléments pédagogiques méritent d'être soulignés dans ce texte : (i) l'avertissement explicite selon lequel la correction **peut contenir des erreurs** ; (ii) l'invitation faite à l'étudiant lui-même à **confronter l'évaluation à d'autres LLM**, transformant l'IA en un outil d'étude actif plutôt qu'en un verdict passif ; et (iii) l'explication transparente du processus automatisé, incluant le ou les modèles utilisés et la politique de retraitement jusqu'à l'obtention d'une note valide.

E.8 Risques, limites éthiques et responsabilité de l'enseignant

! L'enseignant ne peut pas simplement adopter la note de l'IA

C'est le point central de cette annexe : **en aucun cas la note suggérée par l'IA ne doit être automatiquement attribuée à l'étudiant comme note officielle**. Les modèles de langage peuvent halluciner — attribuer des points à des critères non satisfaits, « inventer » qu'une fonction a été appelée correctement alors qu'elle ne l'a pas été, ou pénaliser un code correct en raison d'une mauvaise interprétation de la grille. La note finale de toute activité évaluative doit **toujours** résulter d'une évaluation manuelle de l'enseignant (ou de la correction objective par cas de test, comme dans VPL/MCTest), l'IA jouant tout au plus le rôle de **première lecture** ou de **support pour l'étudiant**, jamais celui d'instance décisionnelle.

Quelques précautions pratiques adoptées dans l'utilisation de **vpl-ai-feedback**, recommandées à tout enseignant souhaitant adapter le système :

- **Séparation claire entre la note officielle et le feedback de l'IA.** Dans le cas rapporté, la note du bonus des simulacres provenait toujours du VPL, jamais de l'IA (Section D.1). La grille de l'IA a été communiquée à l'étudiant comme support d'apprentissage, avec un avertissement explicite indiquant qu'elle peut contenir des erreurs.
- **Transparence envers l'étudiant.** L'e-mail envoyé (Section D.6) précise explicitement que l'évaluation a été générée par IA, indique le ou les modèles utilisés et invite l'étudiant à comparer avec d'autres LLM — réduisant l'asymétrie d'information et encourageant la pensée critique vis-à-vis de la correction reçue.
- **Retraitement plutôt qu'un *cache* invalide.** Le système n'enregistre `rubrica.txt` que lorsqu'il obtient une évaluation valide (Section D.4) ; cela évite qu'une réponse malformée, incomplète ou visiblement incohérente de l'IA soit présentée à l'étudiant comme définitive.
- **Suivi des divergences.** Le rapport CSV consolidé (colonne `Diferenca = Total_IA - Total_Moodle`) permet à l'enseignant d'identifier rapidement les cas où l'IA et la correction objective divergent le plus — comme dans l'exemple de la Section D.5 (+20 points) —, en priorisant ces cas pour une révision manuelle.

- **Canal ouvert pour contestation.** L'e-mail final offre explicitement à l'étudiant la possibilité de signaler des incohérences, l'enseignant restant l'autorité finale sur la note.
- **Données personnelles hors du *prompt*.** Comme décrit à la Section D.2, le nom, l'e-mail, le login, le numéro d'étudiant et le numéro d'identité nationale ne sont jamais envoyés à l'API du LLM, réduisant les risques pour la confidentialité lors de l'utilisation de fournisseurs d'IA externes.

E.9 Remarques finales

vpl-ai-feedback montre comment l'IA peut enrichir le cycle de *feedback* dans les cours de programmation à fort volume de soumissions, en offrant à chaque étudiant une lecture qualitative et individualisée de son propre code — quelque chose d'impossible à faire manuellement pour une centaine d'étudiants, sur plusieurs questions et examens tout au long du trimestre. L'enquête d'opinion citée à la Section D.1 suggère que ce type de retour est bien accueilli par les étudiants. Néanmoins, l'utilisation responsable du système repose sur un principe non négociable, réaffirmé tout au long de cette annexe : **l'IA complète, mais ne remplace pas, l'évaluation de l'enseignant**, et la note officielle de toute activité évaluative doit continuer à être définie par une correction objective (VPL/MCTest, Annexes A et B) et/ou un jugement humain — jamais par la note brute renvoyée par un modèle de langage.

LE PARCOURS DE TNI & VO

SYSTÉMATISER LA CONNAISSANCE, DU PIXEL À L'INTELLIGENCE

Partie I — Fondements de TNI

- 1. Fondements et Premiers Pas
- 2. Échantillonnage, Quantification et Connectivité
- 3. Opérations Spatiales et Filtrage
- 4. Morphologie Mathématique et Segmentation
- 5. Transformations et Compression

Partie II — Vision par Ordinateur

- 6. Analyse de Documents et Inspection
- 7. Classification et Reconnaissance de Motifs
- 8. Compréhension de Scènes et Segmentation
- 9. Apprentissage Profond (Deep Learning)

Annexes Essentielles

- A : MCTest
- B : Moodle/VPL
- C : SEB/Simulateurs
- D : Gemini/Matériel Didactique
- E : IA dans le retour VPL-IA
- F : Perception et Évaluation

IMAGE D'ENTRÉE (PIXEL) → TNI (TRAITEMENT) → VO (VISION) → SORTIE INTELLIGENTE (SÉMANTIQUE)

MAÎTRISEZ LES PIXELS, CONSTRUISEZ L'AVENIR.

Du simple histogramme à la compréhension complexe de scènes grâce au deep learning, cet ouvrage offre un parcours complet. Transformez des images en données et des données en décisions intelligentes.

Votre capacité à façonner la perception par ordinateur commence ici.

Allez de l'avant, explorez et innovez !

Publié au format numérique (HTML, PDF et Jupyter Notebook), avec des simulateurs et des exercices avec des cas de test compatibles avec le VPL de Moodle.

https://fzampirolli.github.io/tni-vo/

ACCÉDEZ À PLUS DE CONTENU SUPPLÉMENTAIRE